

# Database Access

This section describes various aspects of accessing data in a database with Natural. It covers the following topics:

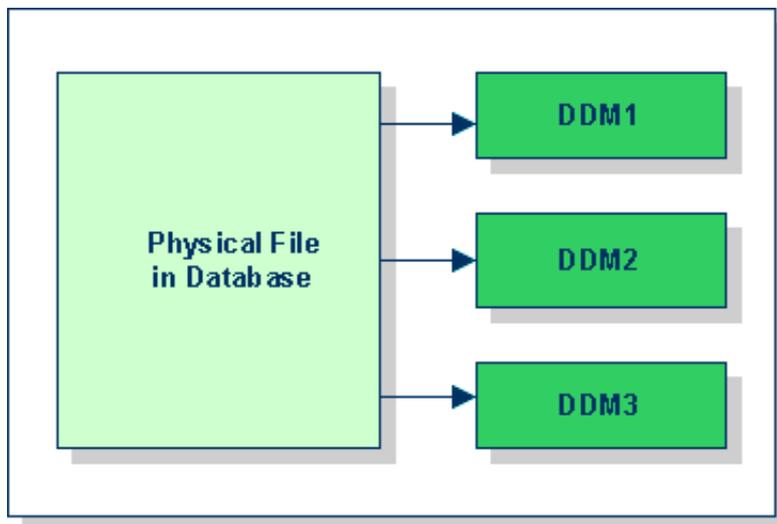
- DDMs (Data Definition Modules)
  - Database Arrays
  - DEFINE DATA Views
  - Statements for Database Access
  - READ Statement
  - FIND Statement
  - HISTOGRAM Statement
  - Database Processing Loops
  - Database Update - Transaction Processing
  - Statements ACCEPT and REJECT
  - AT START/END OF DATA Statements
- 

## DDMs (Data Definition Modules)

For Natural to be able to access a database file, a logical definition of the physical database file is required. Such a logical file definition is called a DDM (data definition module).

The DDM contains information about the individual fields of the file - information which is relevant for the use of these fields in a Natural program. A DDM constitutes a logical view of a physical database file.

For each physical file of a database, one or more DDMs can be defined.



DDMs are defined by the Natural administrator with Predict (or, if Predict is not available, with the corresponding Natural function, as described in your Natural User's Guide for Mainframes documentation).

For each database field, a DDM contains the database-internal field name as well as the "external" field name, that is, the name of the field as used in a Natural program. Moreover, the formats and lengths of the fields are defined in the DDM, as well as various specifications that are used when the fields are output with a DISPLAY or WRITE statement (column headings, edit masks, etc.).

The following topics are covered below:

- Displaying a DDM
- Components of a DDM

## Displaying a DDM

If you do not know the name of the DDM you want, you can use the system command **LIST DDM** to get a list of all existing DDMs that are available. From the list, you can then select a DDM for display.

To display a DDM whose name you know, you use the system command **LIST DDM *ddm-name***.

For example:

### **LIST DDM EMPLOYEES**

A list of all fields defined in the DDM will then be displayed, along with information about each field:

## Components of a DDM

For each field, a DDM contains the following information:

Column	Explanation
<b>T</b>	<p>The <i>type</i> of the field:</p> <p><i>blank</i> Elementary field. This type of field can have only one value within a record.</p> <p><b>M</b> Multiple-value field. This type of field can have more than one value within a record.</p> <p><b>P</b> Periodic group. A periodic group is a group of fields that can have more than one occurrence within a record.</p> <p><b>G</b> Group. A group is a number of fields defined under one common group name. This makes it possible to reference several fields collectively by using the group name instead of the names of all the individual fields.</p> <p><b>*</b> Comment line.</p>
<b>L</b>	<p>The <i>level</i> number assigned to the field.</p> <p>Levels are used to indicate the structure and grouping of the field definitions. This is relevant with view definitions, redefinitions and field groups.</p>
<b>DB</b>	<p>The two-character database-<i>internal field name</i>.</p>
<b>Name</b>	<p>The 3- to 32-character <i>external field name</i>. This is the field name used in a Natural program to reference the field.</p> <p><b>HD=</b> indicates a default column header to appear above the field when the field is output via a DISPLAY statement. If no header is specified, the field name is used as column header.</p> <p><b>EM=</b> indicates a default edit mask to be used when the field is output via a DISPLAY statement.</p>
<b>F</b>	<p>The <i>format</i> of the field (A=alphanumeric, N=numeric unpacked, P=packed numeric, etc.).</p>
<b>Len</b>	<p>The <i>length</i> of the field.</p> <p>For numeric fields, length is specified as "<i>nn.m</i>", where "<i>nn</i>" is the number of digits before the decimal point and "<i>m</i>" is the number of digits after the decimal point.</p>

Column	Explanation
<b>S</b>	<p>The type of <i>suppression</i> assigned to the field:</p> <p><b>N</b> indicates <i>null-value suppression</i>, which means that null values for the field will not be returned when the field is used to construct a basic search criterion (WITH clause of a FIND statement), in a HISTOGRAM statement, or in a READ LOGICAL statement.</p> <p><b>F</b> indicates that the field is defined with the <i>fixed storage</i> option (that is, the field is not compressed).</p> <p>A blank indicates <i>normal compression</i>, which means that trailing blanks in alphanumeric fields and leading zeros in numeric fields are suppressed.</p>
<b>D</b>	<p>The <i>descriptor</i> type of the field; for example:</p> <p><b>D</b> elementary descriptor,  <b>N</b> non-descriptor,  <b>P</b> phonetic descriptor.  <b>U</b> subdescriptor,  <b>S</b> superdescriptor,</p> <p>A blank in this column indicates that the field is not a descriptor.</p> <p>A descriptor can be used as the basis of a database search. A field which has a "D" or "S" in this column can be used in the BY clause of the READ statement. Once a record has been read from the database using the READ statement, a DISPLAY statement can reference any field which has either a "D" or a blank in the "D" column.</p>
<b>Remarks</b>	This column can contain <i>comments</i> about the field.

Above the list of fields, the following is displayed: the number of the file from the DDM is derived (DDM FNR), the number of the database where that file is stored (DDM DBID), and the "Default Sequence" field, that is, the name of the field used to control logical sequential reading of the file if no such field is specified in the READ LOGICAL statement of a program.

## Database Arrays

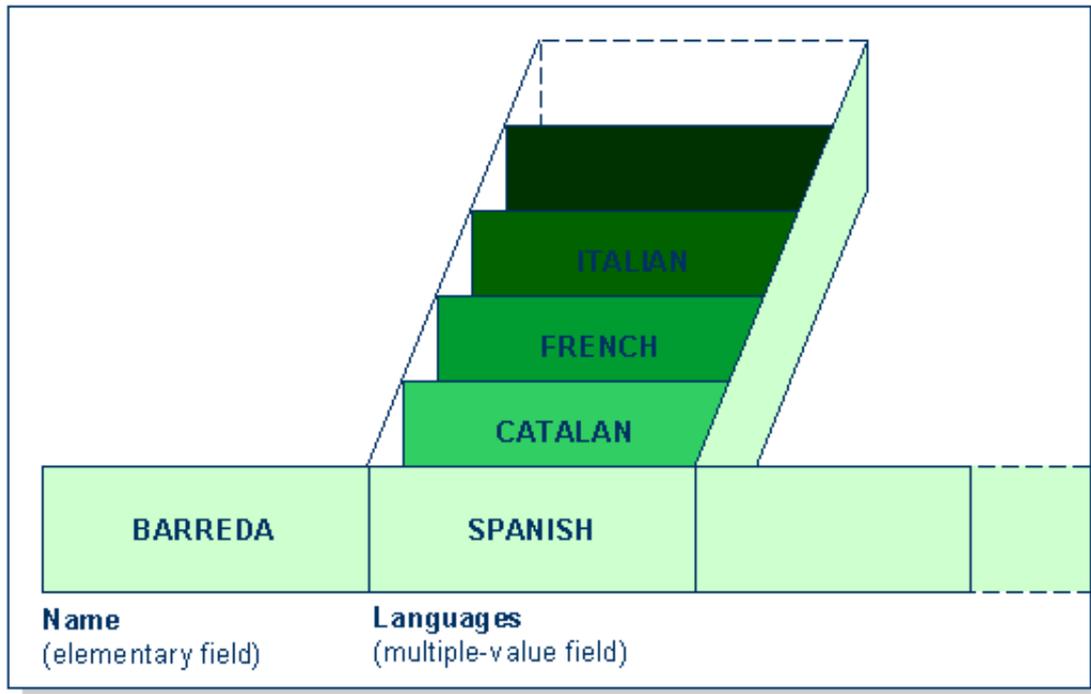
Adabas supports array structures within the database in the form of *multiple-value fields* and *periodic groups*.

- Multiple-Value Fields
- Periodic Groups
- Referencing Multiple-Value Fields and Periodic Groups
- Multiple-Value Fields Within Periodic Groups
- Referencing Multiple-Value Fields Within Periodic Groups
- Referencing the Internal Count of a Database Array

### Multiple-Value Fields

A multiple-value field is a field which can have more than one value (up to 191) within a given record.

#### Example:



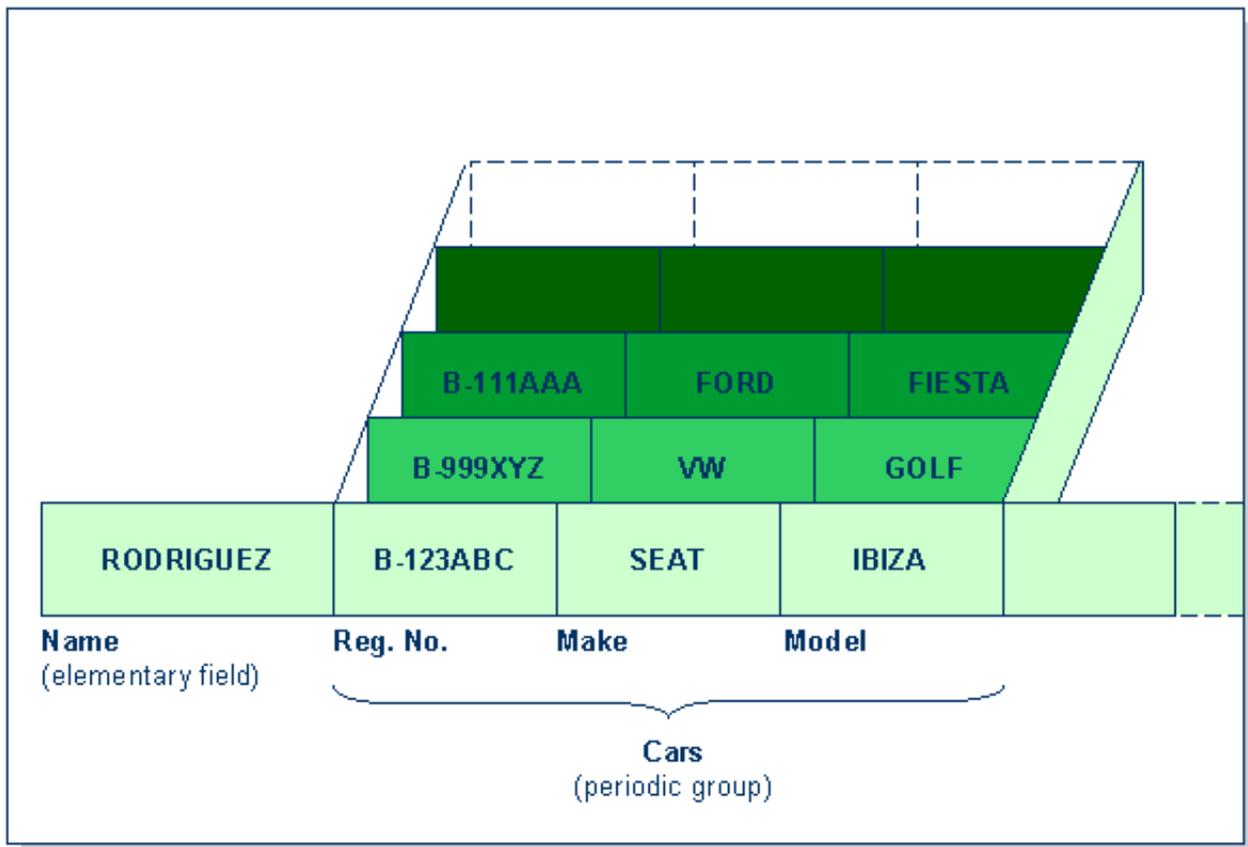
Assuming that the above is a record in an employees file, the first field (Name) is an elementary field, which can contain only one value, namely the name of the person; whereas the second field (Languages), which contains the languages spoken by the person, is a multiple-value field, as a person can speak more than one language.

## Periodic Groups

A periodic group is a group of fields (which may be elementary fields and/or multiple-value fields) that may have more than one occurrence (up to 191) within a given record.

The different values of an multiple-value field are usually called *occurrences*; that is, the number of occurrences is the number of values which the field contains, and a specific occurrence means a specific value. Similarly, in the case of periodic groups, occurrences refer to a group of values.

### Example:



Assuming that the above is a record in a vehicles file, the first field (Name) is an elementary field which contains the name of a person; Cars is a periodic group which contains the automobiles owned by that person. The periodic group consists of three fields which contain the registration number, make and model of each automobile. Each occurrence of Cars contains the values for one automobile.

## Referencing Multiple-Value Fields and Periodic Groups

To reference one or more occurrences of a multiple-value field or a periodic group, you specify an *index notation* after the field name.

### Examples:

The following examples use the multiple-value field LANGUAGES and the periodic group CARS from the previous examples.

The various values of the multiple-value field LANGUAGES can be referenced as follows.

<b>LANGUAGES (1)</b>	References the first value ("SPANISH").
<b>LANGUAGES (X)</b>	The value of the variable X determines the value to be referenced.
<b>LANGUAGES (1:3)</b>	References the first three values ("SPANISH", "CATALAN" and "FRENCH").
<b>LANGUAGES (6:10)</b>	References the sixth to tenth values.
<b>LANGUAGES (X:Y)</b>	The values of the variables X and Y determine the values to be referenced.

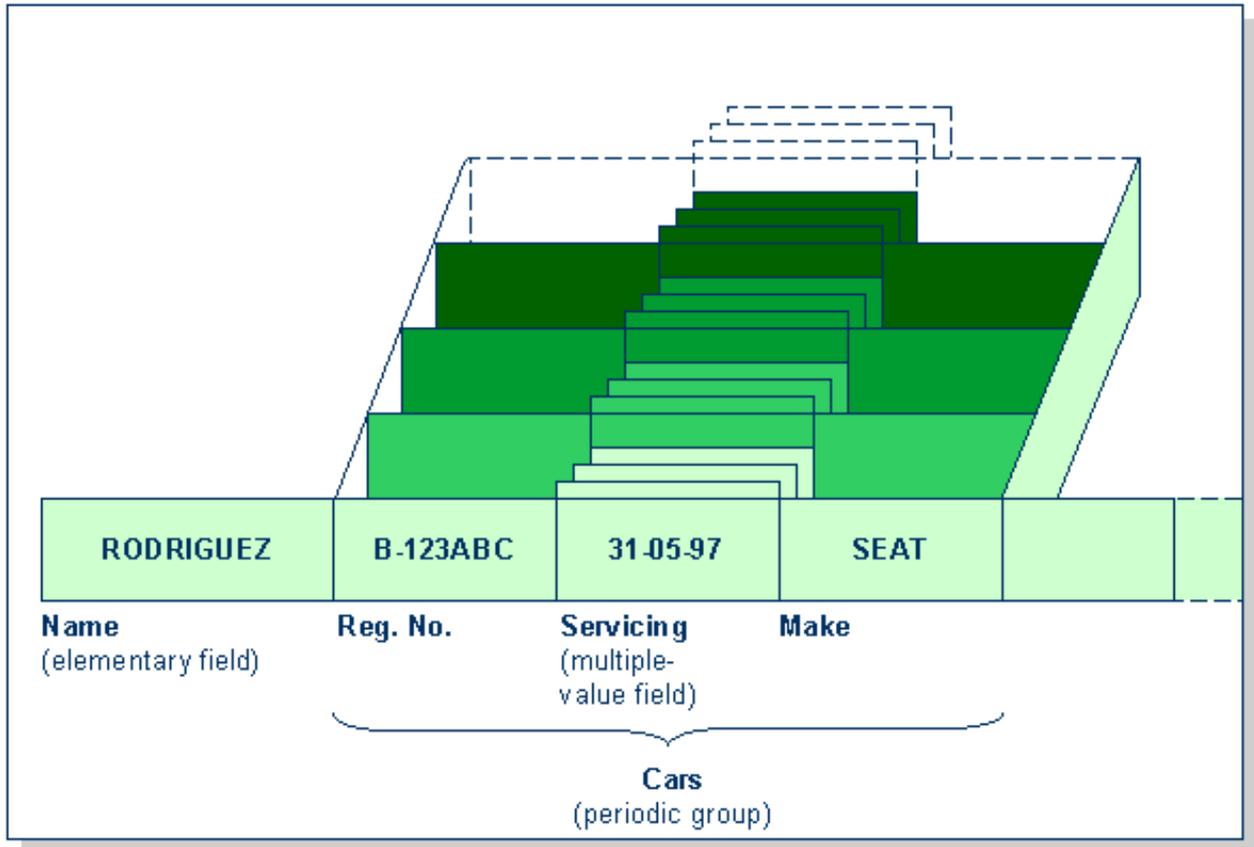
The various occurrences of the periodic group CARS can be referenced in the same manner:

<b>CARS (1)</b>	References the first occurrence ("B-123ABC/SEAT/IBIZA").
<b>CARS (X)</b>	The value of the variable X determines the occurrence to be referenced.
<b>CARS (1:2)</b>	References the first two occurrences ("B-123ABC/ SEAT/IBIZA" and "B-999XYZ/VW/GOLF").
<b>CARS (4:7)</b>	References the fourth to seventh occurrences.
<b>CARS (X:Y)</b>	The values of the variables X and Y determine the occurrences to be referenced.

## Multiple-Value Fields Within Periodic Groups

An Adabas array can have up to two dimensions: a multiple-value field within a periodic group.

### Example:



Assuming that the above is a record in a vehicles file, the first field (Name) is an elementary field which contains the name of a person; Cars is a periodic group which contains the automobiles owned by that person. The periodic group consists of three fields which contain the registration number, servicing dates and make of each automobile. Within the periodic group Cars, the field Servicing is a multiple-value field, containing the different servicing dates for each automobile.

## Referencing Multiple-Value Fields Within Periodic Groups

To reference one or more occurrences of a multiple-value field within a periodic group, you specify a "two-dimensional" index notation after the field name.

### Examples:

The following examples use the multiple-value field **SERVICING** within the periodic group **CARS** from the example above. The various values of the multiple-value field can be referenced as follows:

<b>SERVICING (1,1)</b>	References the first value of <b>SERVICING</b> in the first occurrence of <b>CARS</b> ("31-05-97")
<b>SERVICING (1:5,1)</b>	References the first value of <b>SERVICING</b> in the first five occurrences of <b>CARS</b> .
<b>SERVICING (1:5,1:10)</b>	References the first ten values of <b>SERVICING</b> in the first five occurrences of <b>CARS</b> .

## Referencing the Internal Count of a Database Array

It is sometimes necessary to reference a multiple-value field or a periodic group without knowing how many values/occurrences exist in a given record. Adabas maintains an internal count of the number of values in each multiple-value field and the number of occurrences of each periodic group. This count may be read in a **READ** statement by specifying "C\*" immediately before the field name.

The count is returned in format/length N3. See the Natural Reference documentation for further details.

### Examples:

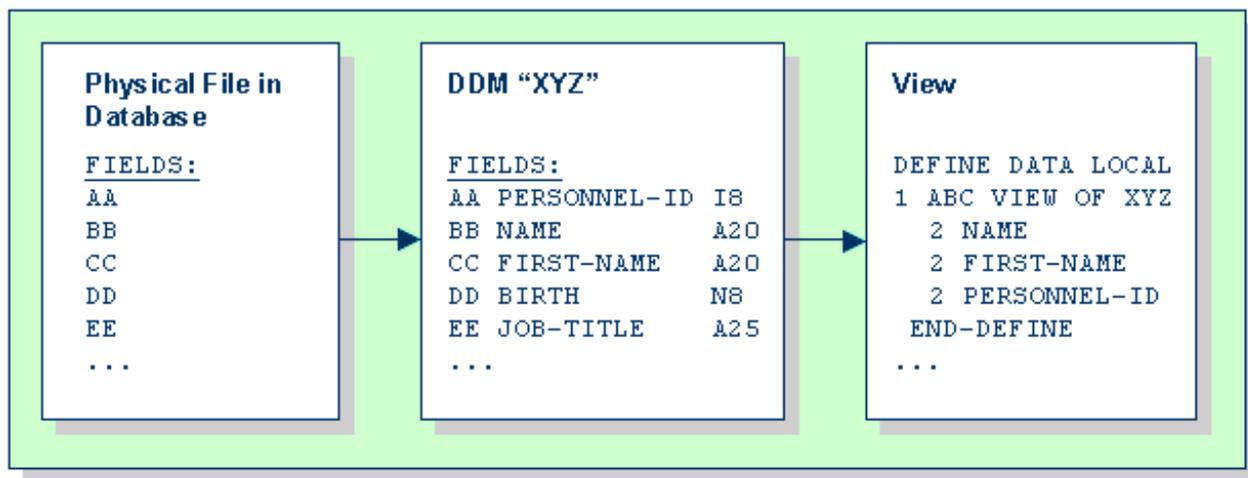
<b>C*LANGUAGES</b>	Returns the number of values of the multiple-value field <b>LANGUAGES</b> .
<b>C*CARS</b>	Returns the number of occurrences of the periodic group <b>CARS</b> .
<b>C*SERVICING(1)</b>	Returns the number of values of the multiple-value field <b>SERVICING</b> in the first occurrence of a periodic group (assuming that <b>SERVICING</b> is a multiple-value field within a periodic group.)

## DEFINE DATA Views

To be able to use database fields in a Natural program, you must specify the fields in a *view*.

In the view, you specify the name of the DDM from which the fields are taken, and the names of the database fields themselves (that is, their long names, not their database-internal short names).

You define such a database view either within the DEFINE DATA statement of the program, or in a local or global data area outside the program with the DEFINE DATA statement referencing that data area (as described in the section Defining Fields).



At level 1, you specify the view name as follows:

**1** *view-name* **VIEW OF** *ddm-name*

where *view-name* is the name you choose for the view, and *ddm-name* is the name of the DDM from which the fields specified in the view are taken. Below that, at level 2, you specify the names of the database fields from the DDM.

In the illustration above, the name of the view is "ABC", and it comprises the fields NAME, FIRST-NAME and PERSONNEL-ID from the DDM "XYZ".

The format and length of a database field need not be specified in the view, as these are already defined in the underlying DDM.

The view may comprise an entire DDM or only a subset of it. The order of the fields in the view need not be the same as in the underlying DDM.

As shown later in this section, the view name is used in database access statements to determine which database is to be accessed.

## Statements for Database Access

To read data from a database, the following statements are available:

- **READ**  
This statement is used to select a range of records from a database in a specified sequence.
- **FIND**  
This statement is used to select from a database those records which meet a specified search criterion.
- **HISTOGRAM**  
This statement is used to read only the values of one database field, or determine the number of records which meet a specified search criterion.

## READ Statement

The READ statement is used to read records from a database. The records can be retrieved from the database:

- in the order in which they are physically stored in the database (READ IN PHYSICAL SEQUENCE), or
- in the order of Adabas Internal Sequence Numbers (READ BY ISN), or
- in the order of the values of a descriptor field (READ IN LOGICAL SEQUENCE).

In this documentation, only READ IN LOGICAL SEQUENCE is discussed, as it is the most frequently used form of the READ statement; for information on the other two options, please refer to the description of the READ statement in the Natural Statements documentation.

The following topics are covered below:

- Syntax
- Limiting the Number of Records to be Read
- The STARTING/ENDING Clauses
- The WHERE Clause

## Syntax

The basic syntax of the READ statement is:

**READ** *view* **IN LOGICAL SEQUENCE BY** *descriptor*

or shorter:

**READ** *view* **LOGICAL BY** *descriptor*

*view* is the name of a view defined in the DEFINE DATA statement (as explained earlier in this section).

*descriptor* is the name of a database field defined in that view. The values of this field determine the order in which the records are read from the database.

If you specify a descriptor, you need not specify the keyword "LOGICAL":

**READ** *view* **BY** *descriptor*

If you do not specify a descriptor, the records will be read in the order of values of the field defined as default descriptor (under "Default Sequence") in the DDM. However, if you specify no descriptor, you must specify the keyword "LOGICAL":

**READ** *view* **LOGICAL**

### Example:

```
** Example Program 'READX01'  
  DEFINE DATA LOCAL  
  1 MYVIEW VIEW OF EMPLOYEES  
    2 PERSONNEL-ID  
    2 NAME  
    2 JOB-TITLE  
  END-DEFINE  
  READ (6) MYVIEW BY NAME  
    DISPLAY NAME PERSONNEL-ID JOB-TITLE  
  END-READ  
  END
```

With the READ statement in the above example, records from the EMPLOYEES file are read in alphabetical order of their last names.

The above program will produce the following output, displaying the information of each employee in alphabetical order of the employees' last names:

Page	1		99-08-19 13:16:04
	NAME	PERSONNEL ID	CURRENT POSITION
	-----	-----	-----
	ABELLAN	60008339	MAQUINISTA
	ACHIESON	30000231	DATA BASE ADMINISTRATOR
	ADAM	50005800	CHEF DE SERVICE
	ADKINSON	20008800	PROGRAMMER
	ADKINSON	20009800	DBA
	ADKINSON	2001100	

If you wanted to read the records to create a report with the employees listed in sequential order by date of birth, the appropriate READ statement would be:

```
READ MYVIEW BY BIRTH
```

You can only specify a field which is defined as a "descriptor" in the underlying DDM (it can also be a subdescriptor, superdescriptor or hyperdescriptor).

## Limiting the Number of Records to be Read

As shown in the previous example program, you can limit the number of records to be read by specifying a number in parentheses after the keyword READ:

```
READ (6) MYVIEW BY NAME
```

In that example, the READ statement would read no more than 6 records.

Without the limit notation, the above READ statement would read *all* records from the EMPLOYEES file in the order of last names from A to Z.

## The STARTING/ENDING Clauses

The READ statement also allows you to qualify the selection of records based on the **value** of a descriptor field. With an EQUAL TO/STARTING from option in the BY or WITH clause, you can specify the value at which reading should begin. By adding a THRU/ENDING AT option, you can also specify the value in the logical sequence at which reading should end.

For example, if you wanted a list of those employees in the order of job titles starting with "TRAINEE" and continuing on to "Z", you would use one of the following statements:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
  READ MYVIEW WITH JOB-TITLE STARTING from 'TRAINEE'
  READ MYVIEW BY JOB-TITLE = 'TRAINEE'
  READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE'
```

Note that the value to the right of the equal sign (=) or STARTING from option must be enclosed in apostrophes. If the value is numeric, this *text notation* is not required.

If a BY option is used, a WITH option cannot be used and vice versa.

The sequence of records to be read can be even more closely specified by adding an end limit with a THRU or ENDING AT clause.

To read just the records with the job title "TRAINEE", you would specify:

```
READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE' THRU 'TRAINEE'
  READ MYVIEW WITH JOB-TITLE EQUAL TO 'TRAINEE'
  ENDING AT 'TRAINEE'
```

To read just the records with job titles that begin with "A" or "B", you would specify:

```
READ MYVIEW BY JOB-TITLE = 'A' THRU 'C'
  READ MYVIEW WITH JOB-TITLE STARTING from 'A' ENDING AT 'C'
```

The values are read up to and including the value specified after THRU/ENDING AT. In the two examples above, all records with job titles that begin with "A" or "B" are read; if there were a job title "C", this would also be read, but not the next higher value "CA".

## The WHERE Clause

The WHERE clause may be used to further qualify which records are to be read.

For instance, if you wanted only those employees with job titles starting from "TRAINEE" who are paid in US currency, you would specify:

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
  WHERE CURR-CODE = 'USD'
```

The WHERE clause can also be used with the BY clause as follows:

```
READ MYVIEW BY NAME
  WHERE SALARY = 20000
```

The WHERE clause differs from the WITH/BY clause in two respects:

- The field specified in the WHERE clause need not be a descriptor.
- The expression following the WHERE option is a logical condition. The following logical operators are possible in a WHERE clause:

<b>EQUAL</b>	<b>EQ</b>	<b>=</b>
<b>NOT EQUAL TO</b>	<b>NE</b>	<b>≠</b>
<b>LESS THAN</b>	<b>LT</b>	<b>&lt;</b>
<b>LESS THAN OR EQUAL TO</b>	<b>LE</b>	<b>&lt;=</b>
<b>GREATER THAN</b>	<b>GT</b>	<b>&gt;</b>
<b>GREATER THAN OR EQUAL TO</b>	<b>GE</b>	<b>&gt;=</b>

The following program illustrates the use of the STARTING from, ENDING AT and WHERE clauses:

```

** Example Program 'READX02'
DEFINE DATA LOCAL
1 MYEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 INCOME (1:2)
  3 CURR-CODE
  3 SALARY
  3 BONUS (1:1)
END-DEFINE
*
READ (3) MYVIEW WITH JOB-TITLE = 'TRAINEE' THRU 'TRAINEE'
      WHERE CURR-CODE (*) = 'USD'
      DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
      SKIP 1
END-READ
END

```

It produces the following output:

NAME CURRENT POSITION	INCOME		
	CURRENCY CODE	ANNUAL SALARY	BONUS
-----	-----	-----	-----
SENKO	USD	23000	0
TRAINEE	USD	21800	0
BANGART	USD	25000	0
TRAINEE	USD	23000	0
LINCOLN	USD	24000	0
TRAINEE	USD	22000	0

#### Further Example of READ Statement:

See program READX03 in library SYSEXPG.

## FIND Statement

The FIND statement is used to select from a database those records which meet a specified search criterion.

The following topics are covered below:

- Syntax
- Limiting the Number of Records to be Processed
- The WHERE Clause
- IF NO RECORDS FOUND Condition

## Syntax

The basic syntax of the FIND statement is:

**FIND RECORDS IN** *view* **WITH** *field* = *value*

or shorter:

**FIND** *view* **WITH** *field* = *value*

*view* is the name of a view defined in the DEFINE DATA statement (as explained earlier in this section).

*field* is the name of a database field defined in that view. You can only specify a *field* which is defined as a "descriptor" in the underlying DDM (it can also be a subdescriptor, superdescriptor, hyperdescriptor or phonetic descriptor).

## Limiting the Number of Records to be Processed

In the same way as with the READ statement, you can limit the number of records to be processed by specifying a number in parentheses after the keyword FIND:

```
FIND (6) RECORDS IN MYVIEW WITH NAME = 'CLEGG'
```

In the above example, only the first 6 records that meet the search criterion would be processed.

Without the limit notation, all records that meet the search criterion would be processed.

### Note:

If the FIND statement contains a WHERE clause (see below), records which are rejected as a result of the WHERE clause are **not** counted against the limit.

## The WHERE Clause

With the WHERE clause of the FIND statement, you can specify an additional selection criterion which is evaluated *after* a record (selected with the WITH clause) has been read and *before* any processing is performed on the record.

### Example of WHERE Clause:

```
** Example Program 'FINDX01'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH CITY = 'PARIS'
      WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
      DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
END
```

Note that in this example only those records which meet the criteria of the WITH clause *and* the WHERE clause are processed in the DISPLAY statement.

CITY	CURRENT POSITION	PERSONNEL ID	NAME
PARIS	INGENIEUR COMMERCIAL	50007300	CAHN
PARIS	INGENIEUR COMMERCIAL	50006500	MAZUY
PARIS	INGENIEUR COMMERCIAL	50004400	VALLY
PARIS	INGENIEUR COMMERCIAL	50002800	BRETON
PARIS	INGENIEUR COMMERCIAL	50001000	GIGLEUX

## IF NO RECORDS FOUND Condition

If no records are found that meet the search criteria specified in the WITH and WHERE clauses, the statements within the FIND processing loop are not executed (for the previous example, this would mean that the DISPLAY statement would not be executed and consequently no employee data would be displayed).

However, the FIND statement also provides an IF NO RECORDS FOUND clause, which allows you to specify processing you wish to be performed in the case that no records meet the search criteria.

### Example of IF NO RECORDS FOUND Clause:

```

** Example Program 'FINDX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FIND MYVIEW WITH NAME = 'BLACKMORE'
  IF NO RECORDS FOUND
    WRITE 'NO PERSON FOUND.'
  END-NOREC
  DISPLAY NAME FIRST-NAME
END-FIND
END

```

The above program selects all records in which the field NAME contains the value "BLACKMORE". For each selected record, the name and first name are displayed. If no record with NAME = 'BLACKMORE' is found on the file, the WRITE statement within the IF NO RECORDS FOUND clause is executed:

Page	1	97-08-19	11:44:00
	NAME	FIRST-NAME	
-----			
NO PERSON FOUND.			

### Further Examples of FIND Statement:

See programs FINDX07, FINDX08, FINDX09, FINDX10 and FINDX11 in library SYSEXPB.

## HISTOGRAM Statement

The HISTOGRAM statement is used to either read only the values of one database field, or determine the number of records which meet a specified search criterion.

The HISTOGRAM statement does not provide access to any database fields other than the one specified in the HISTOGRAM statement.

The following topics are covered below:

- Syntax
- Limiting the Number of Values to be Read
- The STARTING/ENDING Clauses
- The WHERE Clause

### Syntax

The basic syntax of the HISTOGRAM statement is:

**HISTOGRAM VALUE IN** *view* **FOR** *field*

or shorter:

**HISTOGRAM** *view field*

*view* is the name of a view defined in the DEFINE DATA statement (as explained earlier in this section). *field* is the name of the database field defined in that view.

### Limiting the Number of Values to be Read

In the same way as with the READ statement, you can limit the number of values to be read by specifying a number in parentheses after the keyword HISTOGRAM:

```
HISTOGRAM (6) MYVIEW FOR NAME
```

In the above example, only the first 6 values of the field NAME would be read.

Without the limit notation, all values would be read.

### The STARTING/ENDING Clauses

Like the READ statement, the HISTOGRAM statement also provides a STARTING from clause and an ENDING AT (or THRU) clause to narrow down the range of values to be read by specifying a starting value and ending value.

#### Examples:

```
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD'  
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD' ENDING AT 'LANIER'  
HISTOGRAM MYVIEW FOR NAME from 'BLOOM' THRU 'ROESER'
```

### The WHERE Clause

The HISTOGRAM statement also provides a WHERE clause which may be used to specify an additional selection criterion that is evaluated *after* a value has been read and *before* any processing is performed on the value. The field specified in the WHERE clause must be the same as in the main clause of the HISTOGRAM statement.

**Example of HISTOGRAM Statement:**

```

** Example Program 'HISTOX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  END-DEFINE
  *
  LIMIT 8
  HISTOGRAM MYVIEW CITY STARTING FROM 'M'
    DISPLAY NOTITLE CITY 'NUMBER OF/PERSONS' *NUMBER *COUNTER
  END-HISTOGRAM
  END
    
```

CITY	NUMBER OF PERSONS	CNT
MADISON	3	1
MADRID	41	2
MAILLY LE CAMP	1	3
MAMERS	1	4
MANSFIELD	4	5
MARSEILLE	2	6
MATLOCK	1	7
MELBOURNE	2	8

In the above program, the system variables \*NUMBER and \*COUNTER are also evaluated by the HISTOGRAM statement, and output with the DISPLAY statement. \*NUMBER contains the number of database records that contain the last value read; \*COUNTER contains the total number of values which have been read.

## Database Processing Loops

Natural automatically creates the necessary processing loops which are required to process data that have been selected from a database as a result of a FIND, READ or HISTOGRAM statement.

**Example:**

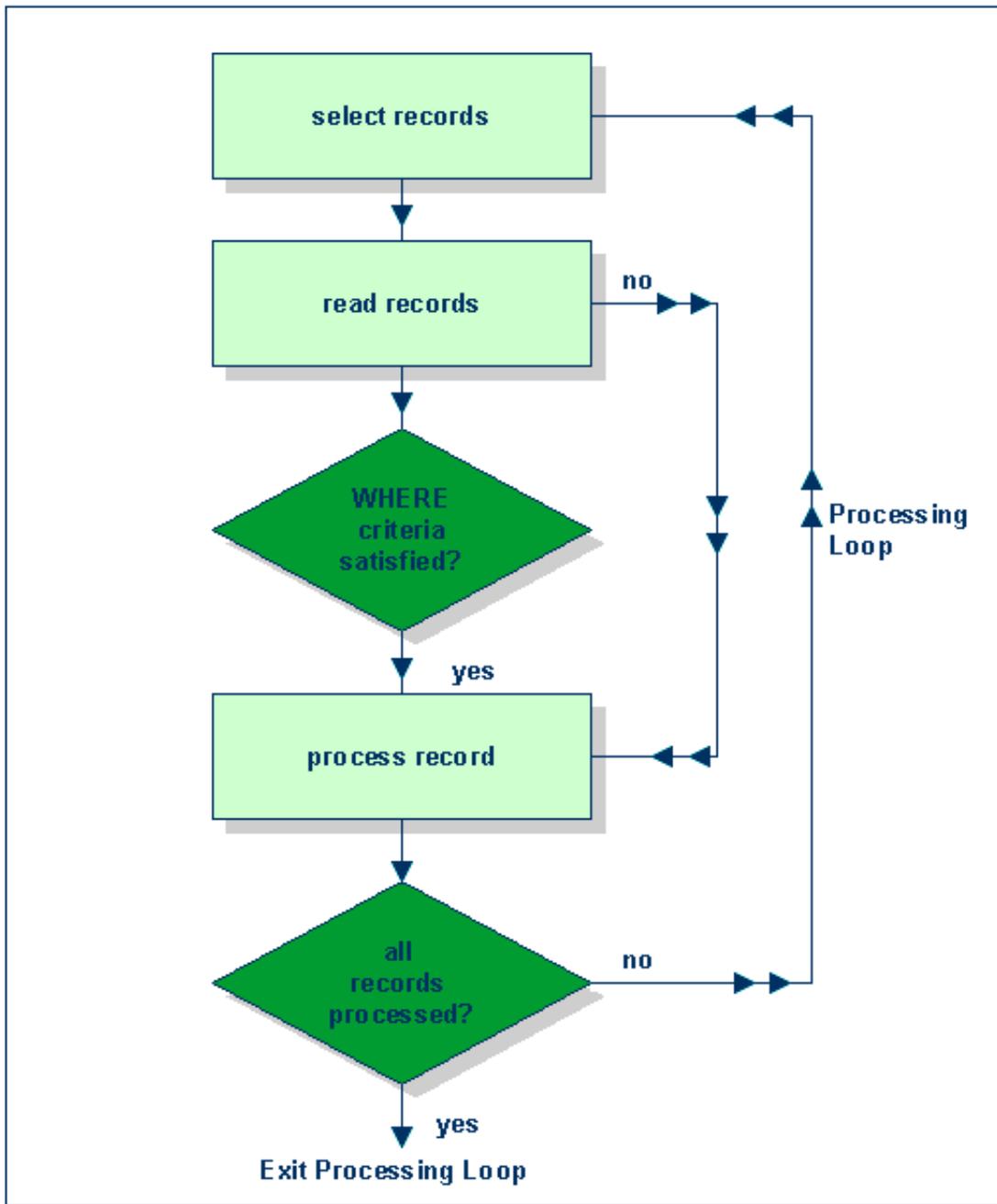
```

** Example Program 'FINDX03'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  END-DEFINE
  *
  FIND MYVIEW WITH NAME = 'ADKINSON'
    DISPLAY NAME FIRST-NAME CITY
  END-FIND
  END
    
```

The above FIND loop selects all records from the EMPLOYEES file in which the field NAME contains the value "ADKINSON" and processes the selected records. In this example, the processing consists of displaying certain fields from each record selected.

If the FIND statement contained a WHERE clause in addition to the WITH clause, only those records that were selected as a result of the WITH clause *and* met the WHERE criteria would be processed.

The following diagram illustrates the flow logic of a database processing loop:



## Hierarchies of Processing Loops

The use of multiple FIND and/or READ statements creates a hierarchy of processing loops, as shown in the following example:

### Example of Processing Loop Hierarchy:

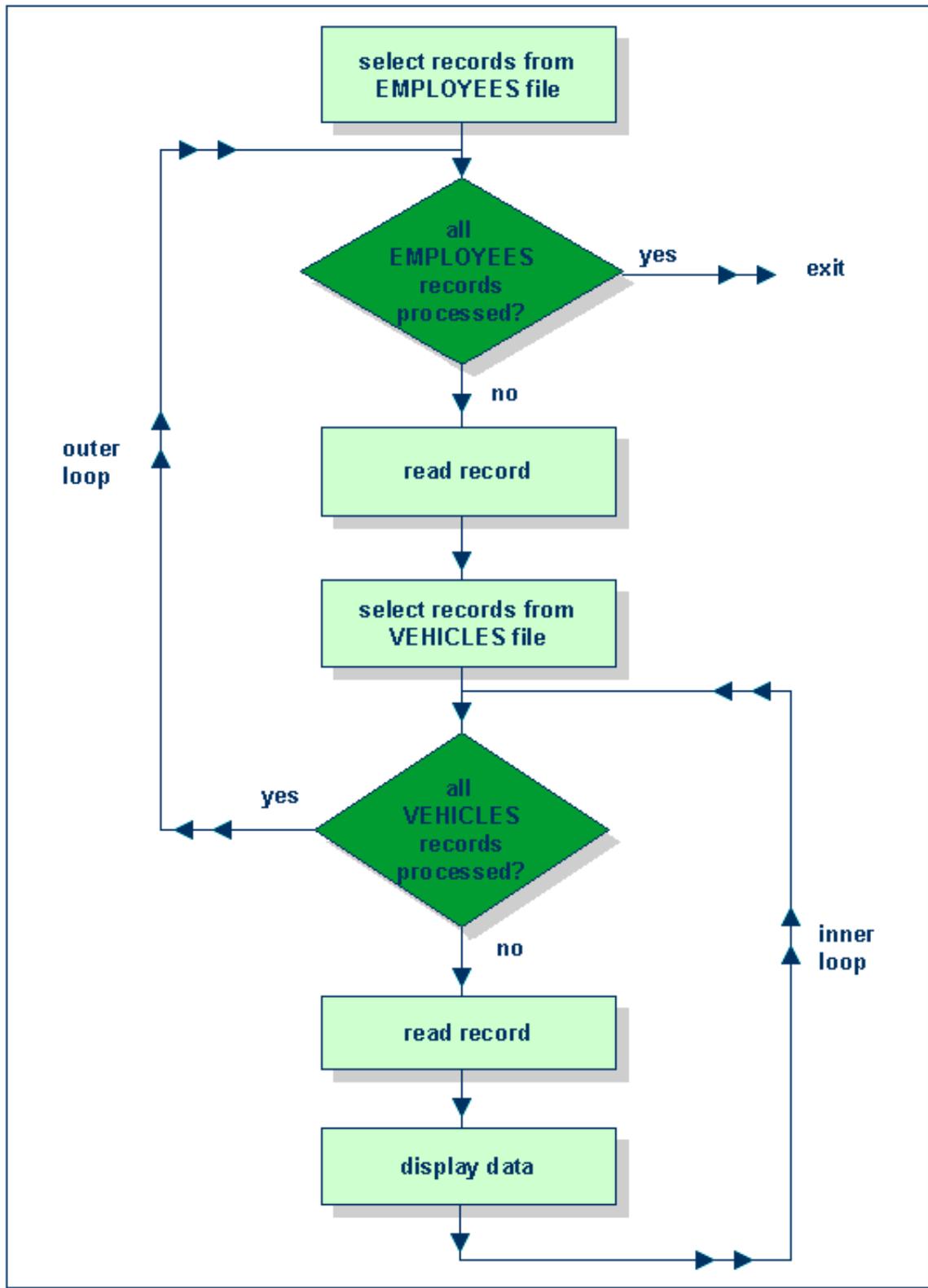
```
** Example Program 'FINDX04'  
DEFINE DATA LOCAL  
1 PERSONVIEW VIEW OF EMPLOYEES  
  2 PERSONNEL-ID  
  2 NAME  
1 AUTOVIEW VIEW OF VEHICLES  
  2 PERSONNEL-ID  
  2 MAKE  
  2 MODEL  
END-DEFINE  
*  
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'  
VEH.  FIND AUTOVIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)  
      DISPLAY NAME MAKE MODEL  
      END-FIND  
      END-FIND  
END
```

The above program selects from the EMPLOYEES file all people with the name "ADKINSON". Each record (person) selected is then processed as follows:

1. The second FIND statement is executed to select the automobiles from the VEHICLES file, using as selection criterion the PERSONNEL-IDs from the records selected from the EMPLOYEES file with the first FIND statement.
2. The NAME of each person selected is displayed; this information is obtained from the EMPLOYEES file. The MAKE and MODEL of each automobile owned by that person is also displayed; this information is obtained from the VEHICLES file.

The second FIND statement creates an inner processing loop within the outer processing loop of the first FIND statement, as shown in the following diagram.

The diagram illustrates the flow logic of the hierarchy of processing loops in the previous example program:



**Example of Nested FIND Loops Accessing the Same File:**

It is also possible to construct a processing loop hierarchy in which the same file is used at both levels of the hierarchy:

```

** Example Program 'FINDX05'
  DEFINE DATA LOCAL
  1 PERSONVIEW VIEW OF EMPLOYEES
    2 NAME
    2 FIRST-NAME
    2 CITY
  1 #NAME (A40)
  END-DEFINE
  *
  WRITE TITLE LEFT JUSTIFIED
    'PEOPLE IN SAME CITY AS:' #NAME / 'CITY:' CITY SKIP 1
  FIND PERSONVIEW WITH NAME = 'JONES'
    WHERE FIRST-NAME = 'LAUREL'
    compress NAME FIRST-NAME INTO #NAME
  FIND PERSONVIEW WITH CITY = CITY
    DISPLAY NAME FIRST-NAME CITY
  END-FIND
  END-FIND
  END

```

The above program first selects all people with name "JONES" and first name "LAUREL" from the EMPLOYEES file. Then all who live in the same city are selected from the EMPLOYEES file and a list of these people is created. All fields values displayed by the DISPLAY statement are taken from the second FIND statement.

PEOPLE IN SAME CITY AS: JONES LAUREL		
CITY: BALTIMORE		
NAME	FIRST-NAME	CITY
-----		
JENSEN	MARTHA	BALTIMORE
LAWLER	EDDIE	BALTIMORE
FORREST	CLARA	BALTIMORE
ALEXANDER	GIL	BALTIMORE
NEEDHAM	SUNNY	BALTIMORE
ZINN	CARLOS	BALTIMORE
JONES	LAUREL	BALTIMORE

**Further Examples of Nested READ and FIND Statements:**

See programs READX04 and LIMITX01 in library SYSEXP.

## Database Update - Transaction Processing

- Logical Transaction
- Record Hold Logic
- Backing Out a Transaction
- Restarting a Transaction

### Logical Transaction

Natural performs database updating operations based on *transactions*, which means that all database update requests are processed in logical transaction units. A logical transaction is the smallest unit of work (as defined by you) which must be performed in its entirety to ensure that the information contained in the database is logically consistent.

A logical transaction may consist of one or more update statements (DELETE, STORE, UPDATE) involving one or more database files. A logical transaction may also span multiple Natural programs.

A logical transaction begins when a record is put on "hold"; Natural does this automatically when the record is read for updating, for example, if a FIND loop contains an UPDATE or DELETE statement.

The end of a logical transaction is determined by an END TRANSACTION statement in the program. This statement ensures that all updates within the transaction have been successfully applied, and releases all records that were put on "hold" during the transaction.

#### Example:

```
DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
FIND MYVIEW WITH NAME = 'SMITH'
  DELETE
  END TRANSACTION
END-FIND
END
```

Each record selected would be put on "hold", deleted, and then - when the END TRANSACTION statement is executed - released from "hold".

#### Note:

The OPRB parameter, as set by the Natural administrator, determines whether or not Natural will generate an END TRANSACTION statement at the end of each Natural program. Ask your Natural administrator for details.

#### Example of STORE Statement:

See program STOREX01 in library SYSEXPG.

## Record Hold Logic

If Natural is used with Adabas, any record which is to be updated will be placed in "hold" status until an END TRANSACTION or BACKOUT TRANSACTION statement is issued or the transaction time limit is exceeded.

When a record is placed in "hold" status for one user, the record is not available for update by another user. Another user who wishes to update the same record will be placed in "wait" status until the record is released from "hold" when the first user ends or backs out his/her transaction.

To prevent users from being placed in wait status, the session parameter WH (Wait Hold) can be used (see the Natural Reference documentation).

When you use update logic in a program, you should consider the following:

- The maximum time that a record can be in hold status is determined by the Adabas transaction time limit (Adabas parameter TT). If this time limit is exceeded, you will receive an error message and all database modifications done since the last END TRANSACTION will be made undone.
- The number of records on hold and the transaction time limit are affected by the size of a transaction, that is, by the placement of the END TRANSACTION statement in the program. Restart facilities should be considered when deciding where to issue an END TRANSACTION. For example, if a majority of records being processed are *not* to be updated, the GET statement is an efficient way of controlling the "holding" of records. This avoids issuing multiple END TRANSACTION statements and reduces the number of ISNs on hold. When you process large files, you should bear in mind that the GET statement requires an additional Adabas call. An example of a GET statement is shown below.

### Example of GET Statement:

```

DEFINE DATA LOCAL
  1 EMPLOY-VIEW VIEW OF EMPLOYEES
    2 NAME
    2 SALARY (1)
END-DEFINE
RD. READ EMPLOY-VIEW BY NAME
  IF SALARY (1) > 30000
    GE.  GET EMPLOY-VIEW *ISN (RD.)
         compute SALARY (1) = SALARY (1) * 1.15
         UPDATE (GE.)
         END TRANSACTION
  END-IF
END-READ
END

```

On mainframe computers, the placing of records in "hold" status is also controlled by the profile parameter RI, as set by the Natural administrator.

## Backing Out a Transaction

During an active logical transaction, that is, before the END TRANSACTION statement is issued, you can cancel the transaction by using a BACKOUT TRANSACTION statement. The execution of this statement removes all updates that have been applied (including all records that have been added or deleted) and releases all records held by the transaction.

## Restarting a Transaction

With the END TRANSACTION statement, you can also store transaction-related information. If processing of the transaction terminates abnormally, you can read this information with a GET TRANSACTION DATA statement to ascertain where to resume processing when you restart the transaction.

### Example of Using Transaction Data to Restart a Transaction:

The following program updates the EMPLOYEES and VEHICLES files. After a restart operation, the user is informed of the last EMPLOYEES record successfully processed. The user can resume processing from that EMPLOYEES record. It would also be possible to set up the restart transaction message to include the last VEHICLES record successfully updated before the restart operation.

```

** Example Program 'GETTRX01'
DEFINE DATA LOCAL
01 PERSON VIEW OF EMPLOYEES
    02 PERSONNEL-ID      (A8)
    02 NAME              (A20)
    02 FIRST-NAME       (A20)
    02 MIDDLE-I         (A1)
    02 CITY              (A20)
01 AUTO VIEW OF VEHICLES
    02 PERSONNEL-ID      (A8)
    02 MAKE              (A20)
    02 MODEL             (A20)
01 ET-DATA
    02 #APPL-ID          (A8) INIT <' '>
    02 #USER-ID          (A8)
    02 #PROGRAM          (A8)
    02 #DATE             (A10)
    02 #TIME             (A8)
    02 #PERSONNEL-NUMBER (A8)
END-DEFINE
*
GET TRANSACTION DATA #APPL-ID #USER-ID #PROGRAM
                    #DATE      #TIME      #PERSONNEL-NUMBER
*
IF #APPL-ID NOT = 'NORMAL' /* IF LAST EXECUTION ENDED ABNORMALLY
AND #APPL-ID NOT = ' '
INPUT (AD=OIL)
    // 20T '*** LAST SUCCESSFUL TRANSACTION ***' (I)
    / 20T '*****'
    /// 25T 'APPLICATION:' #APPL-ID
    / 32T 'USER:' #USER-ID
    / 29T 'PROGRAM:' #PROGRAM
    / 24T 'COMPLETED ON:' #DATE 'AT' #TIME
    / 20T 'PERSONNEL NUMBER:' #PERSONNEL-NUMBER
END-IF
REPEAT
INPUT (AD=MIL) // 20T 'ENTER PERSONNEL NUMBER:' #PERSONNEL-NUMBER
IF #PERSONNEL-NUMBER = 99999999
    ESCAPE bottom
END-IF

```

```

FIND1. FIND PERSON WITH PERSONNEL-ID = #PERSONNEL-NUMBER
IF NO RECORDS FOUND
  REINPUT 'SPECIFIED NUMBER DOES NOT EXIST; ENTER ANOTHER ONE.'
END-NOREC
FIND2. FIND AUTO WITH PERSONNEL-ID = #PERSONNEL-NUMBER
IF NO RECORDS FOUND
  WRITE 'PERSON DOES NOT OWN ANY CARS'
END-NOREC
IF *COUNTER (FIND1.) = 1 /* FIRST PASS THROUGH THE LOOP
  INPUT (AD=M)
    / 20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
    / 20T '-----'
  /// 20T 'NUMBER:' PERSONNEL-ID (AD=O)
    / 22T 'NAME:' NAME ' ' FIRST-NAME ' ' MIDDLE-I
    / 22T 'CITY:' CITY
    / 22T 'MAKE:' MAKE
    / 21T 'MODEL:' MODEL
  UPDATE (FIND1.) /* UPDATE THE EMPLOYEES FILE
ELSE /* SUBSEQUENT PASSES THROUGH THE LOOP
  INPUT NO ERASE (AD=M) ////////// 20T MAKE / 20T MODEL
END-IF
UPDATE (FIND2.) /* UPDATE THE VEHICLES FILE
MOVE *APPLIC-ID TO #APPL-ID
MOVE *INIT-USER TO #USER-ID
MOVE *PROGRAM TO #PROGRAM
MOVE *DAT4E TO #DATE
MOVE *TIME TO #TIME
END TRANSACTION #APPL-ID #USER-ID #PROGRAM
                #DATE #TIME #PERSONNEL-NUMBER
END-FIND /* FOR VEHICLES (FIND2.)
END-FIND /* FOR EMPLOYEES (FIND1.)
END-REPEAT /* FOR REPEAT
STOP /* Simulate abnormal transaction end
END TRANSACTION 'NORMAL '
END

```

## Statements ACCEPT and REJECT

The statements ACCEPT and REJECT are used to select records based on user-specified logical criteria.

The statements ACCEPT and REJECT can be used in conjunction with the database access statements READ, FIND and HISTOGRAM.

### Example of ACCEPT Statement:

```

** Example Program 'ACCEPX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 NAME
    2 JOB-TITLE
    2 CURR-CODE (1:1)
    2 SALARY (1:1)
  END-DEFINE
  READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
    ACCEPT IF SALARY (1) >= 40000
    DISPLAY NAME JOB-TITLE SALARY (1)
  END-READ
  END

```

Page	1	97-08-13	17:26:33
NAME	CURRENT POSITION	ANNUAL SALARY	
ADKINSON	DBA	46700	
ADKINSON	MANAGER	47000	
ADKINSON	MANAGER	47000	
AFANASSIEV	DBA	42800	
ALEXANDER	DIRECTOR	48000	
ANDERSON	MANAGER	50000	
ATHERTON	ANALYST	43000	
ATHERTON	MANAGER	40000	

ACCEPT/REJECT statements allow you to specify logical conditions in addition to those that were specified in WITH and WHERE clauses of the READ statement. The logical condition criteria in the IF clause of an ACCEPT/REJECT statement are evaluated *after* the record has been selected and read.

Logical condition operators include the following (see the Natural Reference documentation for more detailed information):

<b>EQUAL</b>	EQ	:=
<b>NOT EQUAL TO</b>	NE	≠
<b>LESS THAN</b>	LT	<
<b>LESS EQUAL</b>	LE	<=
<b>GREATER THAN</b>	GT	>
<b>GREATER EQUAL</b>	GE	>=

Logical condition criteria in ACCEPT/REJECT statements may also be connected with the Boolean operators AND, OR, and NOT. Moreover, parentheses may be used to indicate logical grouping.

#### Example of ACCEPT Statement with AND Operator:

The following program illustrates the use of the Boolean operator AND in an ACCEPT statement.

```

** Example Program 'ACCEPX02'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY (1:1)
END-DEFINE
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
  ACCEPT IF SALARY (1) >= 40000
    AND SALARY (1) <= 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END

```

#### Example of REJECT Statement with OR Operator:

The following program, which uses the Boolean operator OR in a REJECT statement, produces the same output as the ACCEPT statement above, as the logical operators are reversed.

```

** Example Program 'ACCEPX03'
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY (1:1)
END-DEFINE
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
  REJECT IF SALARY (1) < 40000
    OR SALARY (1) > 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END

```

Page	1		97-08-18 12:21:09
	NAME	CURRENT POSITION	ANNUAL SALARY
	-----	-----	-----
	AFANASSIEV	DBA	42800
	ATHERTON	ANALYST	43000
	ATHERTON	MANAGER	40000

**Further Examples of ACCEPT and REJECT Statements:**

See programs ACCEPX04, ACCEPX05 and ACCEPX06 in library SYSEXP.

## AT START/END OF DATA Statements

### AT START OF DATA Statement

The AT START OF DATA statement is used to specify any processing that is to be performed after the first of a set of records has been read in a database processing loop.

The AT START OF DATA statement must be placed within the processing loop.

If the AT START OF DATA processing produces any output, this will be output *before the first field value*. By default, this output is displayed left-justified on the page.

### AT END OF DATA Statement

The AT END OF DATA statement is used to specify processing that is to be performed after all records for a database processing loop have been processed.

The AT END OF DATA statement must be placed within the processing loop.

If the AT END OF DATA processing produces any output, this will be output *after the last field value*. By default, this output is displayed left-justified on the page.

### Example of AT START OF DATA and AT END OF DATA Statements

The following example program illustrates the use of the statements AT START OF DATA and AT END OF DATA. The system variable \*TIME has been incorporated into the AT START OF DATA statement to display the time of day. The system function OLD has been incorporated into the AT END OF DATA statement to display the name of the last person selected.

```

** Example Program 'ATSTAX01'
  DEFINE DATA LOCAL
  1 MYVIEW VIEW OF EMPLOYEES
    2 CITY
    2 NAME
    2 JOB-TITLE
    2 INCOME (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
  END-DEFINE
  WRITE TITLE 'XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT' /
  READ (3) MYVIEW BY CITY STARTING FROM 'E'
    DISPLAY GIVE SYSTEM FUNCTIONS
      NAME (AL=15) JOB-TITLE (AL=15) INCOME (1)
  AT START OF DATA
    WRITE 'RUN TIME:' *TIME /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
  END-READ
  AT END OF PAGE
    WRITE / 'AVERAGE SALARY:' AVER (SALARY(1))
  END-ENDPAGE
  END

```

The program produces the following output:

XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT				
NAME	CURRENT POSITION	INCOME		
		CURRENCY CODE	ANNUAL SALARY	BONUS
-----				
RUN TIME: 11:18:58.2				
DUYVERMAN	PROGRAMMER	USD	34000	0
PRATT	SALES PERSON	USD	38000	9000
MARKUSH	TRAINEE	USD	22000	0
LAST PERSON SELECTED: MARKUSH				
AVERAGE SALARY:		31333		

**Further Examples of AT START OF DATA and AT END OF DATA Statements:**

See programs ATENDX01, ATSTAX02 and WRITEX09 in library SYSEXP.