

Daten in einer Adabas-Datenbank aufrufen

Dieses Dokument beschreibt verschiedene Aspekte des Aufrufs von Daten in einer Adabas-Datenbank mit Natural.

Folgende Themen werden behandelt:

- Adabas-Datenbankverwaltungsschnittstellen ADA und ADA2
 - Datendefinitionsmodule (DDMs)
 - Datenbank-Arrays
 - Datenbank-View definieren
 - Statements für Datenbankzugriffe
 - Multi-Fetch-Klausel
 - Datenbank-Verarbeitungsschleifen
 - Datenänderungen - Transaktionsverarbeitung
 - Datensätze mit ACCEPT/REJECT auswählen
 - AT START/END OF DATA-Statements
 - Unicode-Daten
-

Adabas-Datenbankverwaltungsschnittstellen ADA und ADA2

Bei den in Natural vorhandenen Adabas-Datenbankverwaltungsschnittstellentypen ADA und ADA2 handelt es sich um Schnittstellen, die voneinander so verschieden sind wie zum Beispiel die Typen ADA und SQL.

Der Typ ADA ist die Standardschnittstelle zu Adabas-Datenbanken. Sie kommt in Frage, wenn keine neue Adabas-Funktionalität zu berücksichtigen ist, die mit Adabas Version 6 für Open Systems und Adabas Version 8 für Großrechner eingeführt wurde.

Der Typ ADA2 ist eine erweiterte Schnittstelle zu Adabas-Datenbanken ab Adabas Version 6 für Open Systems und ab Adabas Version 8 für Großrechner. Sie unterstützt insbesondere Adabas-LA-Felder, große Adabas-Objektfelder und erweiterte Adabas-Puffer-Größen. Voraussetzung für die Unterstützung von Adabas-LA-Feldern und großen Objektfeldern ist die Verwendung des Natural-Formats (A) DYNAMIC in einer View-Definition. Die Unterstützung von erweiterten Adabas-Puffer-Größen ermöglicht die Definition von View-Größen von mehr als 64 KB. Weitere Informationen siehe *Datenbank-View definieren*.

Der Datenbanktyp ADA2 unterstützt keine Multi-Fetch-Verarbeitung. Entsprechende globale und lokale Definitionen werden zur Laufzeit ignoriert.

Software AG-Produkte, die eine eigene Systemdatei besitzen, benötigen eine entsprechende physische Datenbank des Datenbanktyps ADA.

Natural-Objekte, die mit dem Datenbanktyp ADA kompiliert wurden, können in einer Umgebung ausgeführt werden, in der die entsprechende Datenbank als Datenbank vom Typ ADA2 definiert ist.

Datendefinitionsmodule (DDMs)

Damit Natural auf eine Datenbank-Datei zugreifen kann, ist eine logische Definition der physischen Datenbank-Datei erforderlich. Eine solche logische Dateidefinition wird DDM (Datendefinitionsmodul) genannt.

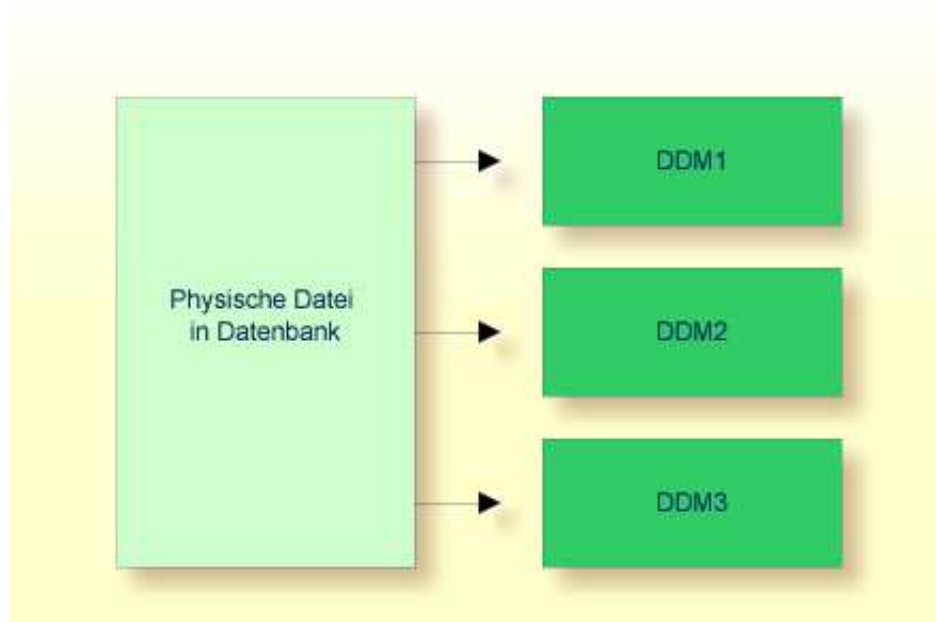
Dieser Abschnitt behandelt folgende Themen:

- Datendefinitionsmodule benutzen
- DDMs verwalten
- DDMs auflisten/anzeigen

Datendefinitionsmodule benutzen

Das DDM enthält Informationen über die einzelnen Felder der Datei — Informationen, die bei der Verwendung dieser Felder in einem Natural-Programm relevant sind. Ein DDM stellt eine logische Sicht (View) auf eine physische Datenbank-Datei dar.

Für jede physische Datei einer Datenbank können ein oder mehrere DDMs definiert werden. Und für jedes DDM können ein oder mehrere Datensichten definiert werden (siehe *View-Definition* in der DEFINE DATA-Statement-Dokumentation).



DDMs werden vom Natural-Administrator mit Predict definiert (oder, falls Predict nicht vorhanden ist, mit der entsprechenden Natural-Funktion zum Verwalten von DDMs).

DDMs verwalten

Ein DDM enthält die datenbankinternen Feldnamen der Datenbankfelder sowie ihre "externen" Feld-Langnamen (d.h. die in einem Natural-Programm verwendeten Feldnamen). Außerdem sind im DDM Format und Länge der Felder definiert, sowie weitere Angaben, die verwendet werden, wenn ein Feld in einem DISPLAY- oder WRITE-Statement benutzt wird (Spaltenüberschriften, Editiermasken usw.).

Informationen zu den in einem DDM definierten Feldattributen entnehmen Sie dem Abschnitt *Using the DDM Editor Windows* unter *DDM Editor* in der *Natural Editors*-Dokumentation.

DDMs auflisten/anzeigen

Wenn Sie den Namen einer benötigten DDM Nicht kennen, können Sie sich mit dem Systemkommando LIST VIEW eine Liste aller in der aktuellen Library vorhandenen Views (d.h. DDMs) anzeigen lassen. Aus der Liste können Sie dann eine DDM zur Anzeige auswählen.

LIST VIEW	Zeigt eine Liste aller Views (DDMs).
LIST VIEW <i>view-name</i>	Wenn Sie einen einzelnen View-Namen angeben, wird die angegebene View angezeigt. Für den <i>view-name</i> können Sie Stern-Notation (*) oder einen bestimmten Bereich von Views angeben (zum Beispiel: A*).

Datenbank-Arrays

Adabas unterstützt Array-Strukturen innerhalb der Datenbank in Form von *multiplen Feldern* und *Periodengruppen*.

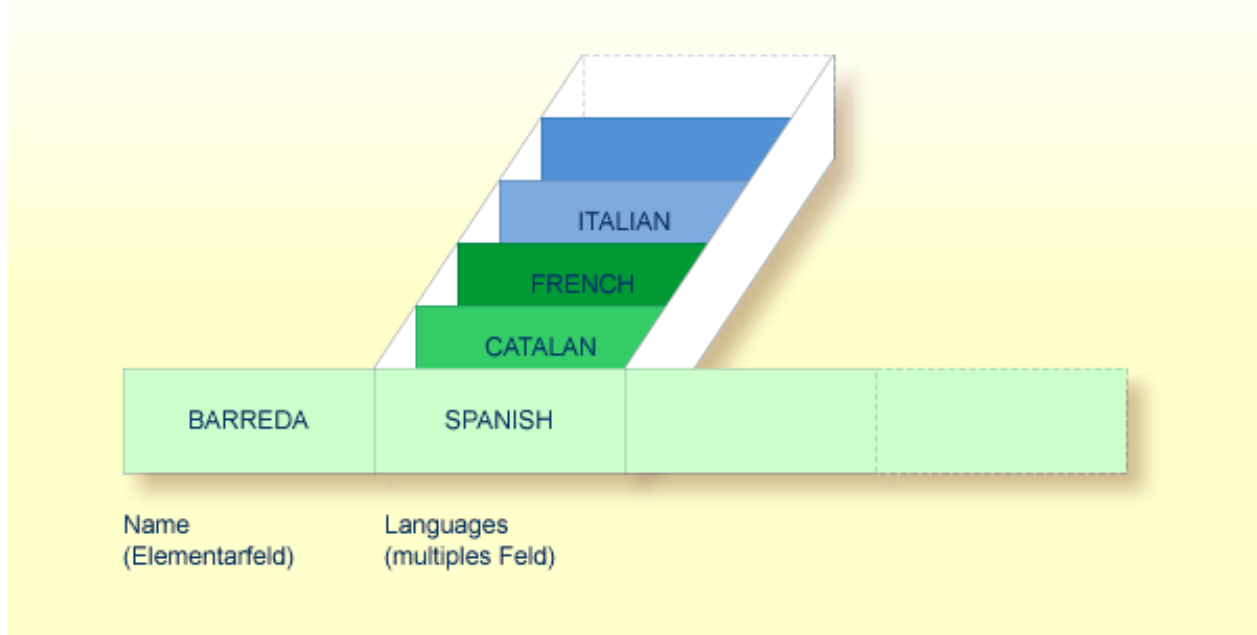
Dieser Abschnitt behandelt folgende Themen:

- Multiple Felder
- Periodengruppen
- Multiple Felder und Periodengruppen referenzieren
- Multiple Felder innerhalb von Periodengruppen
- Multiple Felder innerhalb von Periodengruppen referenzieren
- Internen Zähler eines Datenbank-Arrays referenzieren

Multiple Felder

Ein *multiple Feld* ist ein Feld, das innerhalb eines Datensatzes mehr als einen Wert (bis zu 191) haben kann.

Beispiel:



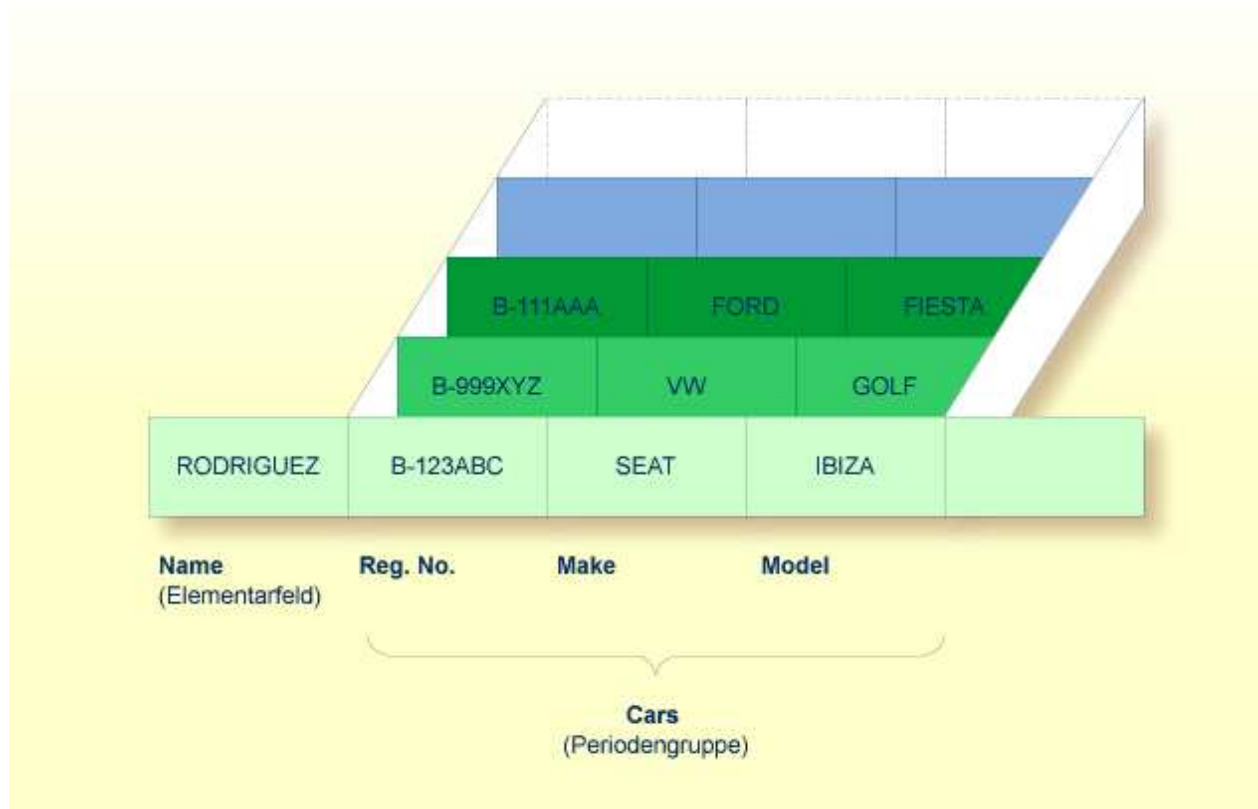
Angenommen, obige Abbildung zeigt einen Datensatz aus einer Personendatei: das erste Feld (Name) ist ein Elementarfeld, das nur einen Wert enthalten kann, nämlich den Namen der Person; das zweite Feld (Languages) enthält die Sprachen, die die Person spricht, und ist ein multiples Feld, da eine Person mehrere Sprachen sprechen kann.

Periodengruppen

Eine *Periodengruppe* ist eine Gruppe von Feldern (wobei es sich um Elementarfelder und/oder multiple Felder handeln kann), die innerhalb eines Datensatzes mehr als eine Ausprägung (bis zu 191) haben kann.

Bei multiplen Feldern werden die verschiedenen Werte eines Feldes auch als *Ausprägungen* bezeichnet, d.h. mit der Anzahl der Ausprägungen ist die Anzahl der Werte, die das Feld enthält, gemeint, und eine bestimmte Ausprägung bezeichnet einen bestimmten Wert. Analog dazu ist bei einer Periodengruppe mit Ausprägung eine Gruppe von Werten gemeint.

Beispiel:



Angenommen, obige Abbildung zeigt einen Datensatz aus einer Fahrzeugdatei: das erste Feld (Name) ist ein Elementarfeld, das den Namen einer Person enthält; Cars ist eine Periodengruppe, die die Fahrzeuge dieser Person enthält. Die Periodengruppe besteht aus drei Feldern, die für jedes Fahrzeug das KFZ-Kennzeichen (Reg. No.), die Marke (Make) und das Modell (Model) enthalten. Jede Ausprägung von Cars enthält jeweils die Werte für ein Fahrzeug.

Multiple Felder und Periodengruppen referenzieren

Um eine oder mehrere Ausprägungen eines multiplen Feldes oder einer Periodengruppe zu referenzieren, geben Sie hinter dem Feldnamen eine *Index-Notation* an.

Beispiele:

Die folgenden Beispiele verwenden das multiple Feld LANGUAGES und die Periodengruppe CARS aus den obigen Abbildung.

Die verschiedenen Werte des multiplen Feldes LANGUAGES können wie folgt referenziert werden:

LANGUAGES (1)	Referenziert den ersten Wert (SPANISH).
LANGUAGES (X)	Der Inhalt der Variablen X bestimmt den zu referenzierenden Wert.
LANGUAGES (1:3)	Referenziert die ersten drei Werte (SPANISH, CATALAN und FRENCH).
LANGUAGES (6:10)	Referenziert den sechsten bis zehnten Wert.
LANGUAGES (X:Y)	Die Inhalte der Variablen X und Y bestimmen die zu referenzierenden Werte.

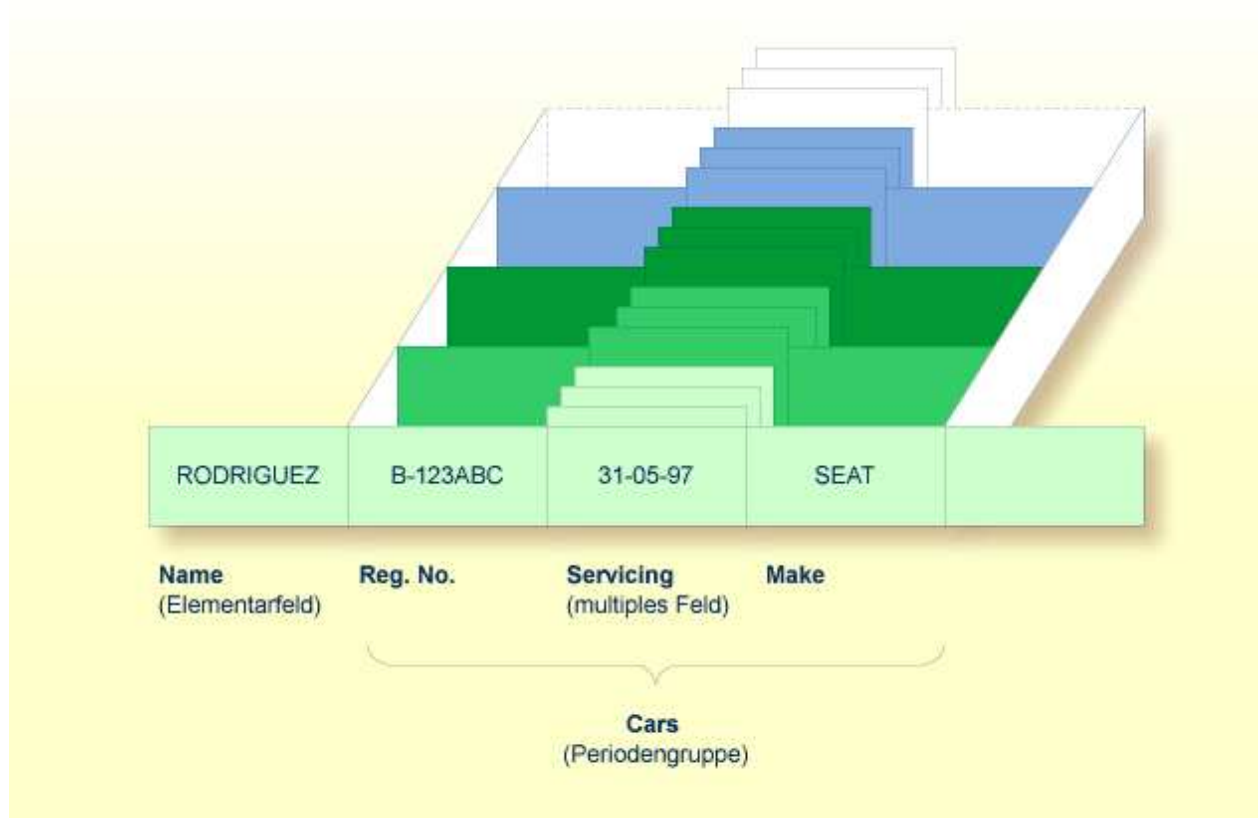
Die verschiedenen Ausprägungen der Periodengruppe CARS können in der gleichen Weise referenziert werden:

CARS (1)	Referenziert die erste Ausprägung (B-123ABC/SEAT/IBIZA).
CARS (X)	Der Inhalt der Variablen X bestimmt die zu referenzierende Ausprägung.
CARS (1:2)	Referenziert die ersten beiden Ausprägungen (B-123ABC/SEAT/IBIZA und B-999XYZ/VW/GOLF).
CARS (4:7)	Referenziert die vierte bis siebte Ausprägung.
CARS (X:Y)	Die Inhalte der Variablen X und Y bestimmen die zu referenzierenden Ausprägungen.

Multiple Felder innerhalb von Periodengruppen

Ein Adabas-Array kann bis zu zwei Dimensionen haben: ein multiples Feld innerhalb einer Periodengruppe.

Beispiel:



Angenommen, obige Abbildung zeigt einen Datensatz aus einer Fahrzeugdatei: das erste Feld (Name) ist ein Elementarfeld, das den Namen einer Person enthält; Cars ist eine Periodengruppe, die die Fahrzeuge dieser Person enthält. Die Periodengruppe besteht aus drei Feldern, die für jedes Fahrzeug das KFZ-Kennzeichen (Reg. No.), die Inspektionstermine (Servicing) und die Marke (Make) enthalten. Innerhalb der Periodengruppe Cars ist Servicing ein multiples Feld, das die verschiedenen

Inspektionstermine jedes Autos enthält.

Multiple Felder innerhalb von Periodengruppen referenzieren

Um eine oder mehrere Ausprägungen eines multiplen Feldes innerhalb einer Periodengruppe zu referenzieren, geben Sie eine "zweidimensionale" Index-Notation hinter dem Feldnamen an.

Beispiele:

Die folgenden Beispiele verwenden das multiple Feld `SERVICING` und die Periodengruppe `CARS` aus der obigen Abbildung. Die verschiedenen Werte des multiplen Feldes können wie folgt referenziert werden:

<code>SERVICING (1,1)</code>	Referenziert den ersten Wert von <code>SERVICING</code> in der ersten Ausprägung von <code>CARS</code> (31-05-97).
<code>SERVICING (1:5,1)</code>	Referenziert jeweils den ersten Wert von <code>SERVICING</code> in den ersten fünf Ausprägungen von <code>CARS</code> .
<code>SERVICING (1:5,1:10)</code>	Referenziert jeweils die ersten zehn Werte von <code>SERVICING</code> in den ersten fünf Ausprägungen von <code>CARS</code> .

Internen Zähler eines Datenbank-Arrays referenzieren

Es ist manchmal erforderlich, ein multiples Feld oder eine Periodengruppe zu referenzieren, ohne die Anzahl der Werte bzw. Ausprägungen eines Datensatzes zu kennen. Adabas zählt intern die Anzahl der Werte eines multiplen Feldes und die Anzahl der Ausprägungen einer Periodengruppe. Dieser interne Zähler kann mit einem `READ`-Statement abgelesen werden, indem man unmittelbar vor dem Feldnamen `C*` angibt:

Die Anzahl wird jeweils in Format/Länge N3 zurückgegeben. Weitere Informationen entnehmen Sie dem Abschnitt *Internen Zähler eines Datenbank-Arrays referenzieren* in der *Statements*-Dokumentation.

Beispiele:

<code>C*LANGUAGES</code>	Liefert die Anzahl der Werte des multiplen Feldes <code>LANGUAGES</code> .
<code>C*CARS</code>	Liefert die Anzahl der Ausprägungen der Periodengruppe <code>CARS</code> .
<code>C*SERVICING(1)</code>	Liefert die Anzahl der Werte des multiplen Feldes <code>SERVICING</code> in der ersten Ausprägung einer Periodengruppe (ausgehend von der Annahme, dass <code>SERVICING</code> ein multiples Feld innerhalb einer Periodengruppe ist).

Datenbank-View definieren

Um Datenbankfelder in einem Natural-Programm verwenden zu können, müssen Sie sie in einer sogenannten View (Datenbanksicht) angeben.

In dem View geben Sie Folgendes an: den Namen des Datendefinitionsmoduls (siehe *Datendefinitionsmodule (DDMs)*), aus dem die Felder stammen, und die Namen der Datenbankfelder selbst (d.h. ihre Langnamen, nicht ihre datenbankinternen Kurznamen).

Ein View kann ein komplettes DDM umfassen oder einen Ausschnitt daraus. Die Reihenfolge der Felder in der View braucht nicht mit der Reihenfolge der Felder im zugrundeliegenden DDM übereinzustimmen.

Wie im Abschnitt *Statements für Datenbankzugriffe* noch gezeigt wird, wird der View-Name in den Statements READ, FIND, HISTOGRAM verwendet, um zu bestimmen, auf welche Datenbank zugegriffen werden soll.

Weitere Informationen bezüglich der vollständigen Syntax der View-Definition oder über die Definition/Redefinition einer Gruppe von Feldern siehe *View-Definition* in der Beschreibung des DEFINE DATA-Statements in der *Statements-Dokumentation*.

Sie haben folgende Möglichkeiten, um eine Datenbank-View zu definieren:

- **Innerhalb des Programms**

Sie können eine Datenbank-View innerhalb des Programms, d.h. direkt im DEFINE DATA-Statement des Programms definieren.

- **Außerhalb des Programms**

Sie können eine Datenbank-View außerhalb des Programms, d.h. in einem separaten Programmierobjekt definieren: entweder in einer Local Data Area (LDA) oder in einer Global Data Area (GDA), wobei das DEFINE DATA-Statement dann diese Data Area referenziert.

▶ **Um eine Datenbank-View innerhalb des Programms zu definieren**

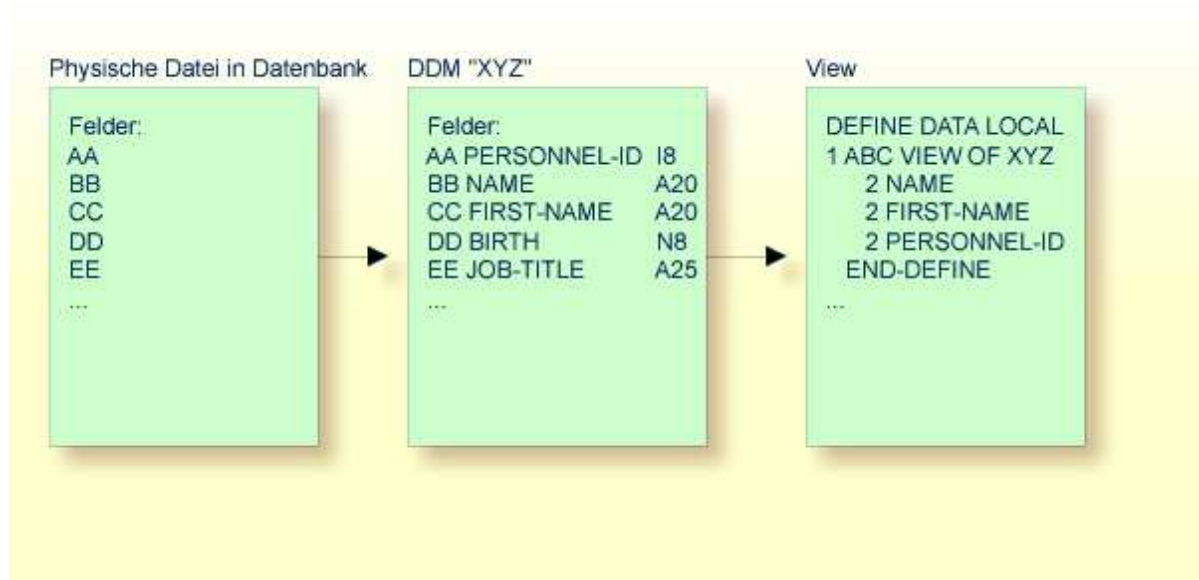
1. Auf Level 1 geben Sie den View-Namen wie folgt an:

```
1 view-name VIEW OF ddm-name
```

wobei *view-name* der von Ihnen gewählte Name für den View ist, und *ddm-name* der Name des DDMs, aus dem die im View angegebenen Felder stammen.

2. Darunter, auf Level 2, geben Sie die Namen der Datenbankfelder aus dem DDM an.

In der folgenden Abbildung hat die View den Namen ABC und umfasst die Felder NAME, FIRST-NAME und PERSONNEL-ID aus dem DDM XYZ.



Format und Länge eines Datenbankfeldes brauchen in der View nicht angegeben zu werden, da sie bereits im zugrundeliegenden DDM definiert sind.

Beispiel-Programm:

In diesem Beispiel lautet der View-Name VIEWEMP, der DDM-Name ist EMPLOYEES und die Namen der aus dem DDM stammenden Felder lauten NAME, FIRST-NAME und PERSONNEL-ID.

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

► Um eine Datenbank-View außerhalb des Programms zu definieren

1. Im Programm selbst geben Sie an:

```
DEFINE DATA LOCAL
  USING <data-area-name>
END-DEFINE
...
```

wobei *data-area-name* der von Ihnen gewählte Name für die Local or Global Data Area ist, zum Beispiel LDA39.

2. In der im Programm referenzierten Data Area geben Sie Folgendes an:

1. Auf Level 1 in der Spalte Name den Namen, den Sie für die View gewählt haben, und in der Spalte Miscellaneous den Namen des DDM, aus dem die in der View angegebenen Felder stammen.

2. Darunter, auf Level 2, geben Sie die Namen der Datenbankfelder aus dem DDM an.

Beispiel-Data-Area LDA39:

In diesem Beispiel lautet der View-Name VIEWEMP, der DDM-Name ist EMPLOYEES und die Namen der aus dem DDM stammenden Felder lauten PERSONNEL-ID, FIRST-NAME und NAME.

I T L	Name	F Length	Miscellaneous
All	----->	----->	----->
V	1 VIEWEMP		EMPLOYEES
	2 PERSONNEL-ID	A 8	
	2 FIRST-NAME	A 20	
	2 NAME	A 20	
	1 #VARI-A	A 20	
	1 #VARI-B	N 3.2	
	1 #VARI-C	I 4	

Anmerkungen zu Datenbanken des Typs ADA2

Bei Datenbanken des Typs ADA2 (Angabe erfolgt in der Tabelle DBMS Assignments in der Configuration Utility, siehe *Database Management System Assignments* in der *Configuration Utility* Dokumentation), gilt Folgendes:

- Wenn Felder mit großen alphanumerischen Variablen (LA) oder großen Objekten (LOB) verwendet werden sollen (Adabas LA/LB-Option), können diese Felder in der View-Definition sowohl mit festen Format/Längen-Werten, zum Beispiel A20 or U20, als auch mit dynamischen Format/Länge-Werten, zum Beispiel (A) DYNAMIC or U(DYNAMIC), angegeben werden.
- Außerdem können Längen-Indikator-Felder L@ . . . in Views angegeben werden, wenn sie sich auf LA- bzw. LB-Felder beziehen.

Statements für Datenbankzugriffe

Um Daten von einer Datenbank zu lesen, stehen folgende Statements zur Verfügung:

READ	Mit diesem Statement können Sie eine Reihe von Datensätzen in einer bestimmten Reihenfolge von der Datenbank lesen.
FIND	Mit diesem Statement können Sie von einer Datenbank diejenigen Datensätze lesen, die ein bestimmtes Suchkriterium erfüllen.
HISTOGRAM	Mit diesem Statement können Sie nur die Werte eines einzelnen Datenbankfeldes lesen oder herausfinden, wieviele Datensätze ein bestimmtes Suchkriterium erfüllen.

Das READ-Statement

Folgende Themen werden behandelt:

- Verwendung des READ-Statements
- Syntax-Grundform des READ-Statements
- Beispiel für READ-Statement:
- Anzahl der zu lesenden Datensätze begrenzen

- STARTING- und ENDING-Klausel beim READ-Statement
- WHERE-Klausel beim READ-Statement
- Weiteres Beispiel für READ-Statement

Verwendung des READ-Statements

Das READ-Statement dient dazu, Datensätze von einer Datenbank zu lesen. Die Datensätze können von der Datenbank gelesen werden:

- in der Reihenfolge, in der sie physisch auf der Datenbank gespeichert sind (`READ IN PHYSICAL SEQUENCE`) oder
- in der Reihenfolge der Adabas-internen Satznummern (`READ BY ISN`) oder
- in logischer Reihenfolge der Werte eines Deskriptorfeldes (`READ IN LOGICAL SEQUENCE`).

In diesem Handbuch wird lediglich `READ IN LOGICAL SEQUENCE` behandelt, da dies die am häufigsten verwendete Form des READ-Statements ist.

Informationen zu den anderen beiden Möglichkeiten finden Sie unter der Beschreibung des READ-Statements in der *Statements*-Dokumentation.

Syntax-Grundform des READ-Statements

Die Grundform des READ-Statements ist:

```
READ view IN LOGICAL SEQUENCE BY descriptor
```

oder kürzer:

```
READ view LOGICAL BY descriptor
```

- dabei ist

<i>view</i>	der Name eines im <code>DEFINE DATA</code> -Statement definierten Views (wie im Abschnitt <i>Datenbank-View definieren</i> beschrieben).
<i>descriptor</i>	der Name eines in diesem View definierten Datenbankfeldes. Die Werte dieses Feldes bestimmen die Reihenfolge, in der die Datensätze von der Datenbank gelesen werden.

Wenn Sie einen Deskriptor angeben, erübrigt sich die Angabe des Schlüsselwortes `LOGICAL`:

```
READ view BY descriptor
```

Wenn Sie keinen Deskriptor angeben, werden die Datensätze in der Reihenfolge der Werte des im DDM als Standard-Deskriptor (unter "Default Sequence") definierten Feldes gelesen. Wenn Sie keinen Deskriptor angeben, müssen Sie allerdings das Schlüsselwort `LOGICAL` angeben:

READ view LOGICAL

Beispiel für READ-Statement:

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

Ausgabe des Programms READX01:

Das READ-Statement im obigen Beispiel liest Datensätze von der Mitarbeiter-Datei EMPLOYEES in alphabetischer Reihenfolge der (im Feld NAME enthaltenen) Nachnamen.

Das obige Programm erzeugt folgende Ausgabe, wobei die Informationen zu jedem Mitarbeiter in alphabetischer Reihenfolge der Nachnamen angezeigt werden:

```
Page          1                               04-11-11  14:15:54

      NAME                PERSONNEL          CURRENT
                        ID                   POSITION
-----
ABELLAN                60008339  MAQUINISTA
ACHIESON                3000231  DATA BASE ADMINISTRATOR
ADAM                    50005800  CHEF DE SERVICE
ADKINSON                20008800  PROGRAMMER
ADKINSON                20009800  DBA
ADKINSON                2001100
```

Falls Sie die Mitarbeiterdaten in der Reihenfolge der (im Feld BIRTH enthaltenen) Geburtsdaten lesen und ausgeben möchten, wäre dazu folgendes READ-Statement geeignet:

```
READ MYVIEW BY BIRTH
```

Sie können nur ein Feld angeben, das im zugrundeliegenden DDM als *Deskriptor* definiert ist (es kann auch ein Subdeskriptor, Superdeskriptor, Hyperdeskriptor, phonetischer Deskriptor oder Nicht-Deskriptor sein).

Anzahl der zu lesenden Datensätze begrenzen

Wie im Beispielprogramm auf der vorigen Seite gezeigt, können Sie die Anzahl der Datensätze, die gelesen werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort READ in Klammern eine Zahl angeben:

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

In diesem Beispiel würde das READ-Statement maximal 6 Datensätze lesen.

Ohne diese Limit-Notation würde das obige READ-Statement *sämtliche* Datensätze von der EMPLOYEES-Datei in der Reihenfolge der Nachnamen von A bis Z lesen.

STARTING- und ENDING-Klausel beim READ-Statement

Mit dem READ-Statement können Sie das Suchkriterium für die zu lesenden Datensätze durch einen bestimmten *Wert* eines Deskriptorfeldes weiter einschränken. Mit der Option EQUAL TO/STARTING FROM in einer BY bzw. WITH-Klausel können Sie festlegen, ab welchem Wert die Datensätze gelesen werden sollen. Mit der Option THRU/ENDING AT können Sie darüber hinaus bestimmen, bis zu welchem Wert gelesen werden soll.

Wünschen Sie beispielsweise eine Liste aller Mitarbeiter in der Reihenfolge der Tätigkeitsbezeichnungen (JOB-TITLE) von TRAINEE bis Z, würden Sie eines der folgenden Statements verwenden:

```
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'
```

Bitte beachten Sie, dass der Wert hinter dem Gleichheitszeichen (=) bzw. der STARTING FROM-Option in Apostrophen (') stehen muss. Bei einem numerischen Wert ist diese Text-Notation nicht erforderlich.

Es ist nicht möglich, die Optionen BY und WITH gleichzeitig zu verwenden; es ist jeweils nur eine von beiden gestattet.

Durch Angabe einer THRU bzw. ENDING AT-Klausel können Sie darüber hinaus festlegen, bis zu welchem Punkt Datensätze gelesen werden sollen.

Um nur Datensätze mit der Tätigkeitsbezeichnung TRAINEE zu lesen, müssten Sie folgendes angeben:

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

Um alle Datensätze mit Tätigkeitsbezeichnungen, die mit A oder B anfangen, zu lesen, müssten Sie folgendes angeben:

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

Die Werte werden gelesen bis einschließlich des Wertes, der nach THRU/ENDING AT spezifiziert wird. In den beiden obigen Beispielen werden alle Datensätze mit Tätigkeitsbezeichnungen, die mit A oder B anfangen, gelesen; gäbe es eine Tätigkeitsbezeichnung C, würde diese auch gelesen werden, aber nicht der nächsthöhere Wert CA.

WHERE-Klausel beim READ-Statement

Mit einer WHERE-Klausel können Sie ein zusätzliches Suchkriterium angeben.

Zum Beispiel, wenn Sie nur die Datensätze derjenigen Mitarbeiter mit Tätigkeitsbezeichnung TRAINEE, die in US-Währung (USD) bezahlt werden, lesen wollen, dann geben Sie Folgendes an:

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

Die WHERE-Klausel kann auch zusammen mit einer BY-Klausel verwendet werden, zum Beispiel:

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

Die WHERE-Klausel unterscheidet sich in zwei Punkten von einer BY/WITH-Klausel:

- Das in der WHERE-Klausel angegebene Feld muss kein Deskriptor sein.
- In der WHERE-Klausel wird eine logische Bedingung angegeben.

Folgende logische Operatoren können in einer WHERE-Klausel verwendet werden:

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

Das folgende Programm veranschaulicht die Verwendung der Klauseln STARTING FROM, ENDING AT und WHERE:

```
** Example 'READX02': READ (with STARTING, ENDING and WHERE clause)
*****
DEFINE DATA LOCAL
1 MYVIEW 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
STARTING FROM 'TRAINEE' ENDING AT 'TRAINEE'
                WHERE CURR-CODE (*) = 'USD'
    DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
    SKIP 1
END-READ
END
```

Ausgabe des Programms READX02:

NAME		INCOME		
CURRENT				
POSITION	CURRENCY	ANNUAL	BONUS	
	CODE	SALARY		

SENKO	USD	23000	0
TRAINEE	USD	21800	0
BANGART	USD	25000	0
TRAINEE	USD	23000	0
LINCOLN	USD	24000	0
TRAINEE	USD	22000	0

Weiteres Beispiel für READ-Statement

Siehe folgendes Beispiel-Programm:

- *READX03 - READ-Statement*

Das FIND-Statement

Folgende Themen werden behandelt:

- Verwendung des FIND-Statements
- Syntax-Grundform des FIND Statements
- Anzahl der zu verarbeitenden Datensätze begrenzen
- WHERE-Klausel beim FIND-Statement
- Beispiel für FIND-Statement mit WHERE-Klausel:
- IF NO RECORDS FOUND-Bedingung
- Weitere Beispiele zum FIND-Statement

Verwendung des FIND-Statements

Das FIND-Statement dient dazu, Datensätze von einer Datenbank zu lesen, die ein bestimmtes Suchkriterium erfüllen.

Syntax-Grundform des FIND Statements

Die Grundform des FIND-Statements ist:

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

oder kürzer:

```
FIND view WITH field = value
```

- dabei ist

<i>view</i>	der Name eines im DEFINE DATA-Statement definierten Views (wie im Abschnitt <i>Datenbank-View definieren</i> beschrieben).
<i>field</i>	der Name eines in diesem View definierten Datenbankfeldes.

Sie können nur ein *field* angeben, das im zugrundeliegenden DDM als *Deskriptor* definiert ist (es kann auch ein Subdeskriptor, Superdeskriptor, Hyperdeskriptor, phonetischer Deskriptor oder ein Nicht-Deskriptor sein).

Die vollständige Syntax entnehmen Sie der FIND-Statement-Dokumentation.

Anzahl der zu verarbeitenden Datensätze begrenzen

Ähnlich wie beim READ-Statement (siehe oben) können Sie die Anzahl der Datensätze, die verarbeitet werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort FIND in Klammern eine Zahl angeben:

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

In diesem Beispiel würde das FIND-Statement maximal 6 Datensätze verarbeiten.

Ohne diese Limit-Notation würden alle Datensätze, die das Suchkriterium erfüllen, verarbeitet werden.

Anmerkung:

Wenn das FIND-Statement eine WHERE-Klausel enthält (siehe unten), werden Datensätze, die die WHERE-Klausel *nicht* erfüllen, bei der Ermittlung des Limits nicht berücksichtigt.

WHERE-Klausel beim FIND-Statement

Mit der WHERE-Klausel des FIND-Statements können Sie ein zusätzliches Selektionskriterium angeben, das ausgewertet wird, *nachdem* ein (über die WITH-Klausel ausgewählter) Datensatz gelesen wurde und *bevor* der ausgewählte Datensatz weiterverarbeitet wird.

Beispiel für FIND-Statement mit WHERE-Klausel:

```
** Example 'FINDX01': FIND (with WHERE)
*****
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
```

Anmerkung:

Wie Sie sehen, werden in diesem Beispiel nur die Datensätze, die die Kriterien der WITH-Klausel *und* der WHERE-Klausel erfüllen, im DISPLAY-Statement verarbeitet.

Ausgabe des Programms FINDX01:

CITY	CURRENT POSITION	PERSONNEL ID	NAME
PARIS	INGENIEUR COMMERCIAL	50007300	CAHN
PARIS	INGENIEUR COMMERCIAL	50006500	MAZUY
PARIS	INGENIEUR COMMERCIAL	50004700	FAURIE
PARIS	INGENIEUR COMMERCIAL	50004400	VALLY
PARIS	INGENIEUR COMMERCIAL	50002800	BRETON
PARIS	INGENIEUR COMMERCIAL	50001000	GIGLEUX
PARIS	INGENIEUR COMMERCIAL	50000400	KORAB-BRZOZOWSKI

IF NO RECORDS FOUND-Bedingung

Falls keine Datensätze gefunden werden, die die in der WITH- und WHERE-Klausel angegebenen Suchkriterien erfüllen, werden die innerhalb der FIND-Verarbeitungsschleife angegebenen Statements nicht ausgeführt (für das Beispiel auf der vorigen Seite hieße dies, dass das DISPLAY-Statement nicht ausgeführt würde und folglich keine Mitarbeiterdaten angezeigt würden).

Das FIND-Statement bietet jedoch auch eine IF NO RECORDS FOUND-Klausel, in der Sie eine Verarbeitung angeben können, die ausgeführt werden soll für den Fall, dass kein Datensatz die Suchkriterien erfüllt.

Beispiel für FIND-Statement mit IF NO RECORDS FOUND-Bedingung:

```

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

```

Das obige Programm wählt alle Datensätze aus, in denen das Feld NAME den Wert BLACKSMITH enthält. Von jedem ausgewählten Datensatz werden der Name (NAME) und der Vorname (FIRST-NAME) angezeigt. Falls in der Datei kein Datensatz mit NAME = 'BLACKSMITH' gefunden wird, wird das in der IF NO RECORDS FOUND-Klausel angegebene WRITE-Statement ausgeführt:

Ausgabe des Programms FINDX02:

```

Page          1                               04-11-11  14:15:54

          NAME          FIRST-NAME
-----
NO PERSON FOUND.

```

Weitere Beispiele zum FIND-Statement

Siehe die folgenden Beispiel-Programme:

- *FINDX07 - FIND (mit mehreren Klauseln)*
- *FINDX08 - FIND (mit LIMIT)*
- *FINDX09 - FIND (unter Verwendung von *NUMBER, *COUNTER, *ISN)*
- *FINDX10 - FIND (in Kombination mit READ)*
- *FINDX11 - FIND NUMBER (mit *NUMBER)*

Das HISTOGRAM-Statement

Folgende Themen werden behandelt:

- Verwendung des HISTOGRAM-Statements
- Syntax-Grundform des HISTOGRAM-Statements
- Anzahl der zu lesenden Werte begrenzen
- STARTING- und ENDING-Klausel beim HISTOGRAM-STATEMENT
- WHERE-Klausel beim HISTOGRAM-Statement
- Beispiel für HISTOGRAM-Statement

Verwendung des HISTOGRAM-Statements

Das HISTOGRAM-Statement dient dazu, entweder die Werte eines einzelnen Datenbankfeldes zu lesen oder herauszufinden, wieviele Datensätze ein bestimmtes Suchkriterium erfüllen.

Das HISTOGRAM-Statement kann auf keine anderen Datenbankfelder zugreifen als auf das im HISTOGRAM-Statement angegebene Feld.

Syntax-Grundform des HISTOGRAM-Statements

Die Grundform des HISTOGRAM-Statements ist:

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

oder kürzer:

HISTOGRAM *view* **FOR** *field*

- dabei ist

<i>view</i>	der Name eines im DEFINE DATA-Statement definierten Views (wie im Abschnitt <i>Datenbank-View definieren</i> beschrieben).
<i>field</i>	der Name des in diesem View definierten Datenbankfeldes.

Die vollständige Syntax entnehmen Sie der HISTOGRAM-Statement-Dokumentation.

Anzahl der zu lesenden Werte begrenzen

Ähnlich wie beim READ-Statement (siehe oben) können Sie die Anzahl der Werte, die gelesen werden sollen, begrenzen, indem Sie hinter dem Schlüsselwort HISTOGRAM in Klammern eine Zahl angeben:

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

In diesem Beispiel würden nur die ersten 6 Werte des Feldes NAME gelesen.

Ohne diese Limit-Notation würden alle Werte gelesen.

STARTING- und ENDING-Klausel beim HISTOGRAM-STATEMENT

Wie das READ-Statement (siehe oben) bietet auch das HISTOGRAM-Statement eine STARTING FROM-Klausel- und eine ENDING AT bzw. THRU-Klausel, mit denen Sie den Bereich der zu lesenden Werte durch Angabe eines Startwertes und eines Endwertes eingrenzen können.

Beispiele:

```
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'
```

WHERE-Klausel beim HISTOGRAM-Statement

Das HISTOGRAM-Statement bietet außerdem eine WHERE-Klausel, in der Sie ein zusätzliches Selektionskriterium angeben können, das ausgewertet wird, *nachdem* ein Wert gelesen wurde und *bevor* der Wert weiterverarbeitet wird. Das in der WHERE-Klausel angegebene Feld muss dasselbe sein wie das in der Hauptklausel des HISTOGRAM-Statements angegebene.

Beispiel für HISTOGRAM-Statement

```
** Example 'HISTOX01': HISTOGRAM
*****
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
```

In diesem Programm werden mit dem HISTOGRAM-Statement außerdem die Systemvariablen *NUMBER und *COUNTER ausgewertet und mit dem DISPLAY-Statement ausgegeben. *NUMBER enthält die Anzahl der Datensätze, in denen der zuletzt gelesene Wert vorkommt; *COUNTER enthält die Gesamtanzahl der bisher gelesenen Werte.

Ausgabe des Programms HISTOX01:

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

Multi-Fetch-Klausel

Dieser Abschnitt behandelt die *Multi-Fetch*-Datensatz-Retrieval-Funktionalität für Adabas-Datenbanken.

Die Multi-Fetch-Funktionalität wird nur bei Datenbanken des Typs ADA unterstützt, die in der Tabelle DBMS Assignments in der Configuration Utility angegeben werden; siehe *Database Management System Assignments* in der *Configuration Utility*-Dokumentation. Bei Datenbanken des Typs ADA2 wird die Multi-Fetch-Funktionalität nicht unterstützt.

Folgende Themen werden behandelt:

- Sinn und Zweck der Multi-Fetch-Funktion
- Statements mit Multi-Fetch-Verarbeitung
- Bemerkungen zur Multi-Fetch-Benutzung

Sinn und Zweck der Multi-Fetch-Funktion

Im Standardmodus liest Natural mit einem einzigen Datenbank-Aufruf nicht mehrere Datensätze ein, sondern stets nur ein Datensatz pro Fetch-Modus. Diese Art von Betrieb ist solide und stabil, kann aber einige Zeit in Anspruch nehmen, wenn eine große Anzahl von Datenbank-Sätzen verarbeitet wird. Um die Verarbeitungszeit dieser Programme zu verbessern, können Sie die Multi-Fetch-Verarbeitung nutzen.

Standardmäßig verwendet Natural den Single-Fetch-Modus, um Daten aus einer Adabas-Datenbank abzufragen. Der Abfrage-Modus kann mit dem Natural-Profilparameter MFSET konfiguriert werden.

Mit den Werten ON (Multi-Fetch) und OFF (Single-Fetch) wird das Standard-Verhalten festgelegt. Wenn MFSET auf NEVER gesetzt wird, verwendet Natural immer den Single-Fetch-Modus und ignoriert Einstellungen auf Statement-Ebene.

Der Standard-Verarbeitungsmodus kann außerdem auf Statement-Ebene geändert werden.

Statements mit Multi-Fetch-Verarbeitung

Die Multi-Fetch-Verarbeitung wird bei folgenden Statements unterstützt, die keine Datenbank verändernden Zugriffe ausführen:

- FIND

- READ
- HISTOGRAM

Informationen zur Syntax finden Sie in der Beschreibung der MULTI-FETCH-Klausel bei den Statements FIND, READ und HISTOGRAM.

Bemerkungen zur Multi-Fetch-Benutzung

Falls geschachtelte Datenbankabfrageschleifen, die sich auf dieselbe Adabas-Datei beziehen, in einer der inneren Schleifen UPDATE-Statements enthalten, macht Natural bei der Verarbeitung der äußeren Schleifen mit den aktualisierten Werten weiter. Im Multi-Fetch-Modus bedeutet dies, dass eine äußere logische READ-Schleife repositioniert werden muss, wenn eine innere Datenbankabfrageschleife den Wert aktualisiert, der zur Steuerung der Reihenfolge in der äußeren Schleife benutzt wird. Wenn dieser Versuch einen Konflikt für den aktuellen Deskriptor verursacht, wird ein Fehler zurückgegeben. Um dies zu vermeiden, empfiehlt es sich, dass Sie die Multi-Fetch-Funktion in der äußeren Schleife ausschalten.

Generell ist festzustellen, dass die Multi-Fetch-Verarbeitung die Leistung bei Zugriffen auf eine Adabas-Datenbank erhöht. In manchen Fällen kann es dennoch vorteilhaft sein, den Single-Fetch-Modus zur Leistungssteigerung zu verwenden, insbesondere wenn auch Datenbank verändernde Zugriffe ausgeführt werden.

Datenbank-Verarbeitungsschleifen

Dieser Abschnitt erörtert Verarbeitungsschleifen, die zum Abarbeiten von Daten erforderlich sind, welche von einer Datenbank als Ergebnis eines FIND-, READ- oder HISTOGRAM-Statements ausgewählt wurden.

Folgende Themen werden behandelt:

- Erstellung von Datenbank-Verarbeitungsschleifen
- Hierarchien von Verarbeitungsschleifen
- Beispiel für geschachtelte FIND-Schleifen, die dieselbe Datei aufrufen
- Weitere Beispiele für geschachtelte READ- und FIND-Statements

Erstellung von Datenbank-Verarbeitungsschleifen

Natural initiiert automatisch die Schleifen, die zur Verarbeitung von Daten erforderlich sind, die mit einem FIND-, READ- oder HISTOGRAM-Statement von einer Datenbank ausgewählt wurden.

Beispiel:

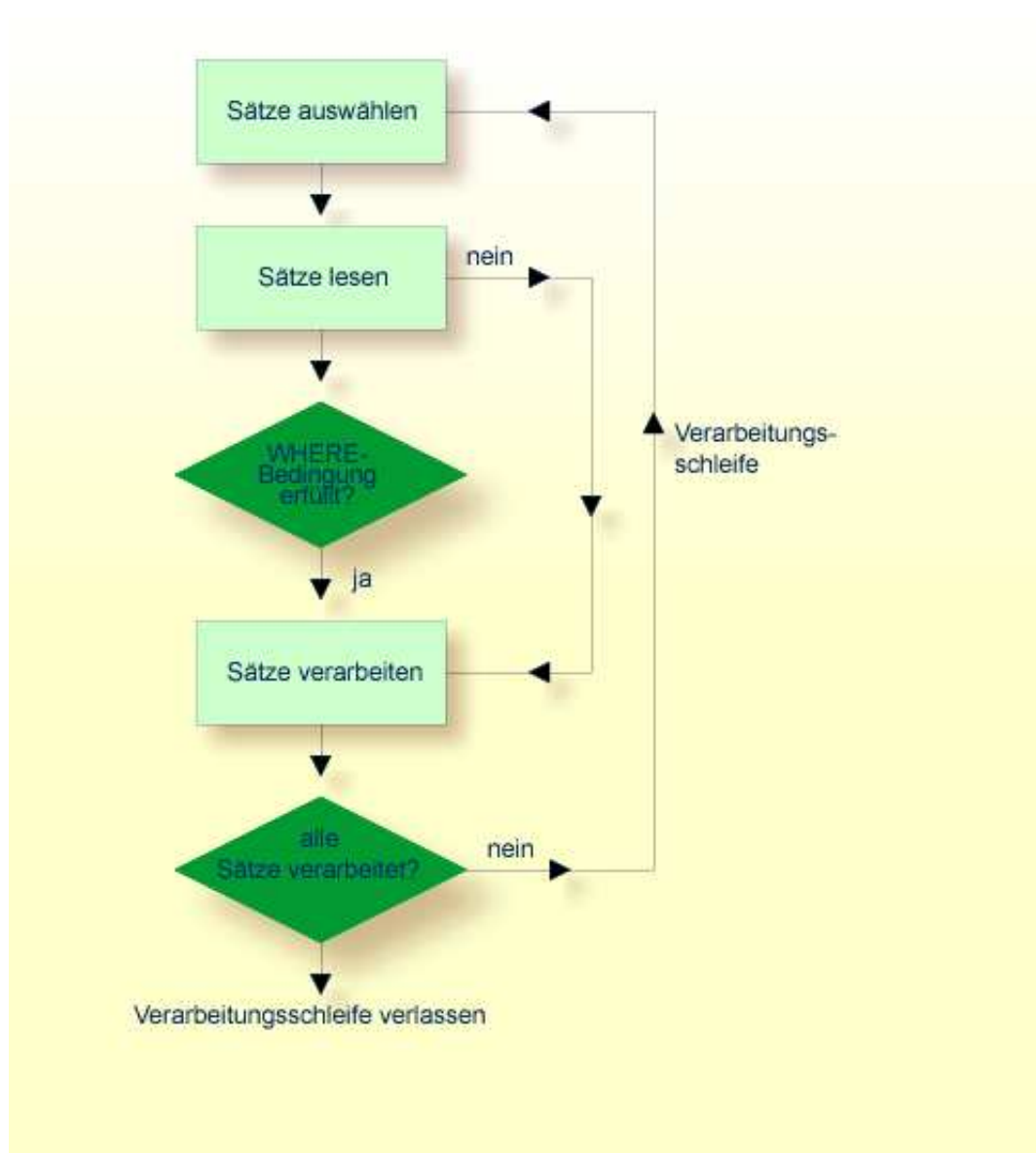
Die obige FIND-Schleife wählt von der Datei EMPLOYEES alle Datensätze aus, in denen das Feld NAME den Wert ADKINSON enthält, und verarbeitet die ausgewählten Datensätze. Im Beispiel besteht die Verarbeitung in der Anzeige bestimmter Feldwerte aus jedem der ausgewählten Datensätze.

```
** Example 'FINDX03': FIND
*****
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
```

Wenn obiges FIND-Statement zusätzlich zu der WITH-Klausel noch eine WHERE-Klausel enthielte, würden nur diejenigen der ausgewählten Datensätze verarbeitet, die die WITH- *und* die WHERE-Bedingung erfüllen.

Das folgende Diagramm zeigt den logischen Ablauf einer Datenbank-Verarbeitungsschleife:



Hierarchien von Verarbeitungsschleifen

Die Verwendung mehrerer FIND- bzw. READ-Statements führt zu einer Hierarchie ineinander geschachtelter Schleifen, wie das folgende Beispiel zeigt:

Beispiel für Verarbeitungsschleifen-Hierarchie:

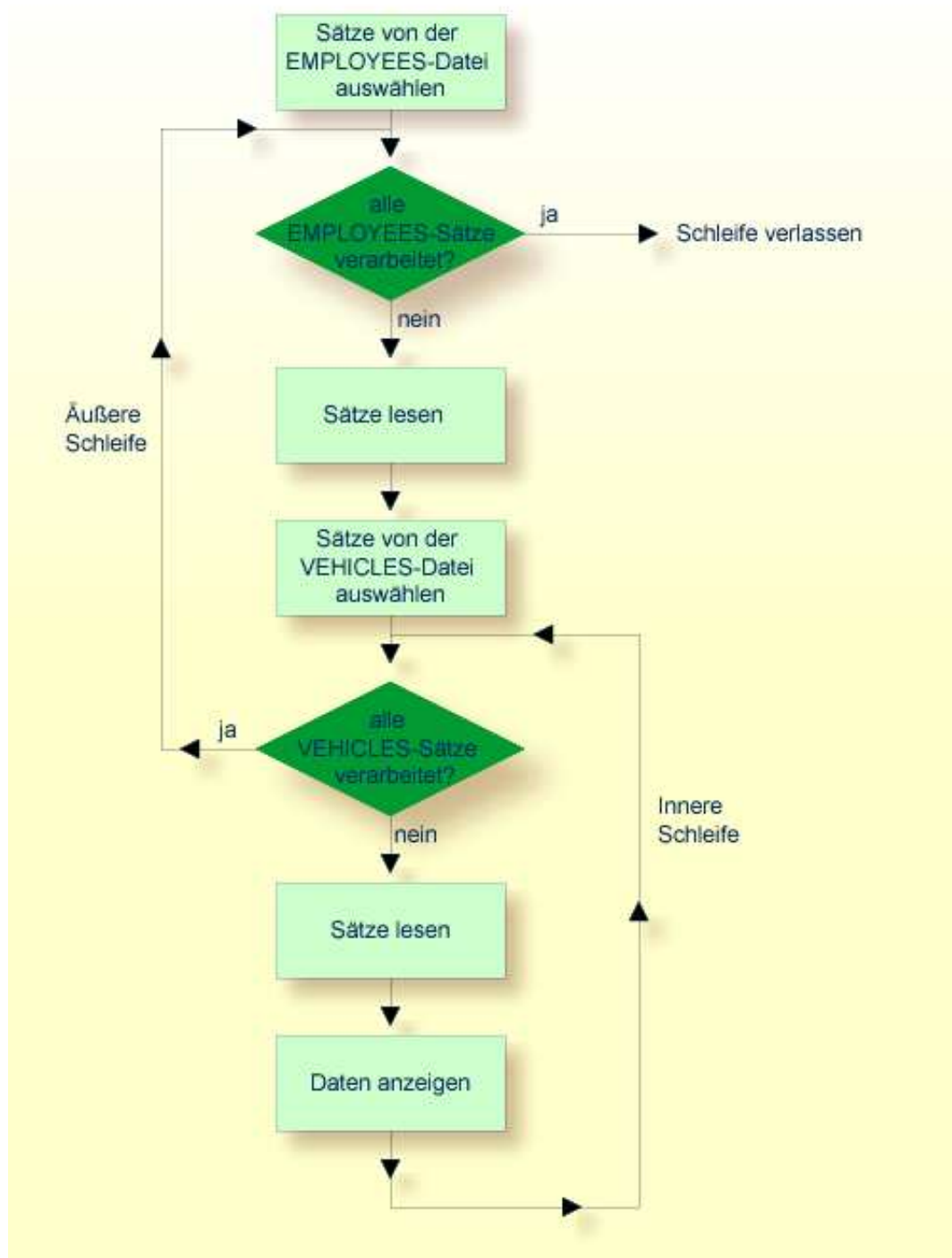
```
** Example 'FINDX04': FIND  (two FIND statements nested)
*****
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 AUTOVIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
  2 MODEL
END-DEFINE
*
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'
  VEH. FIND AUTOVIEW WITH PERSONNEL-ID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

Im obigen Programm werden zunächst alle Datensätze mit Namen ADKINSON von der Datei EMPLOYEES ausgewählt. Dann wird jeder dieser Datensätze (jede Person) wie folgt verarbeitet:

1. Mit dem zweiten FIND-Statement werden für alle von der Datei EMPLOYEES gelesenen Personen die dazugehörigen Fahrzeuge (VEHICLES) gesucht, und zwar unter Verwendung der Personalnummern (PERSONNEL-ID) aus den mit dem ersten FIND-Statement von der EMPLOYEES-Datei ausgewählten Datensätzen.
2. Dann werden mit DISPLAY folgende Werte angezeigt: der NAME jeder gefundenen Person (diese Informationen werden von der EMPLOYEES-Datei gelesen) und Marke und Modell (MAKE und MODEL) des dazugehörigen Fahrzeugs (diese Informationen kommen von der VEHICLES-Datei).

Das zweite FIND-Statement initiiert innerhalb der äußeren FIND-Schleife des ersten FIND-Statements eine innere Schleife, wie das folgende Diagramm veranschaulicht.

Das folgende Diagramm zeigt den logischen Ablauf in der Datenbank-Verarbeitungsschleifen-Hierarchie in dem obigen Beispiel-Programm:



Beispiel für geschachtelte FIND-Schleifen, die dieselbe Datei aufrufen

Es ist auch möglich, eine Verarbeitungsschleifen-Hierarchie aufzubauen, in der zwei ineinander verschachtelte Schleifen auf dieselbe Datei zugreifen, wie das folgende Beispiel zeigt.

```

** Example 'FINDX05': FIND (two FIND statements on same file nested)
*****
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

```

Im obigen Programm werden zunächst in der Datei EMPLOYEES alle Personen mit Namen JONES und Vornamen LAUREL gesucht. Dann werden zu jeder gefundenen Person alle Personen, die in derselben Stadt wohnen, in der EMPLOYEES-Datei gesucht, und es wird eine Liste dieser Personen erzeugt. Alle mit dem DISPLAY-Statement ausgegebenen Feldwerte werden mit dem zweiten FIND-Statement gelesen.

Ausgabe des Programms FINDX05:

```

PEOPLE IN SAME CITY AS: JONES LAUREL
CITY: BALTIMORE

```

NAME	FIRST-NAME	CITY
JENSON	MARTHA	BALTIMORE
LAWLER	EDDIE	BALTIMORE
FORREST	CLARA	BALTIMORE
ALEXANDER	GIL	BALTIMORE
NEEDHAM	SUNNY	BALTIMORE
ZINN	CARLOS	BALTIMORE
JONES	LAUREL	BALTIMORE

Weitere Beispiele für geschachtelte READ- und FIND-Statements

Siehe die folgenden Beispiel-Programme:

- *READX04 - READ-Statement (in Kombination mit FIND und den Systemvariablen *NUMBER und *COUNTER)*
- *LIMITX01 - LIMIT-Statement (für READ- und FIND-Schleifenverarbeitung)*

Datenänderungen - Transaktionsverarbeitung

Dieser Abschnitt beschreibt, wie Natural Datenbankänderungsoperationen mittels Transaktionen durchführt.

Folgende Themen werden behandelt:

- Logische Transaktionen
- Datensatz-Kontrolle während einer Transaktion (Hold-Logik)
- Transaktion abbrechen
- Transaktion neu starten
- Beispiel für Verwendung von Transaktionsdaten beim Neustarten einer Transaktion

Logische Transaktionen

Natural führt Veränderungen auf der Datenbank auf der Grundlage von Transaktionen aus, d.h. alle Veränderungszugriffe werden in logische Transaktionseinheiten gegliedert. Eine Transaktion ist die kleinste (von Ihnen definierte) Verarbeitungseinheit, die vollständig ausgeführt werden muss, um die logische Konsistenz der gespeicherten Daten zu gewährleisten.

Eine logische Transaktion kann aus einem oder mehreren datenverändernden Statements (DELETE, STORE, UPDATE) bestehen und auf eine oder mehrere Dateien zugreifen. Eine logische Transaktion kann sich auch über mehrere Natural-Programme erstrecken.

Eine logische Transaktion beginnt, sobald ein Datensatz in den Hold-Status gestellt wird. Dies erfolgt durch Natural automatisch, wenn ein Satz zwecks Änderung gelesen wird, wenn also z.B. in einer FIND-Schleife ein UPDATE- oder DELETE-Statement steht.

Das Ende einer logischen Transaktion wird im Programm durch ein END TRANSACTION-Statement bestimmt. Dieses Statement gewährleistet, dass alle durch die Transaktion bewirkten Änderungen erfolgreich durchgeführt werden, und gibt anschließend alle während der Transaktion gehaltenen Datensätze wieder frei.

Beispiel:

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

Jeder gefundene Satz würde hier in den Hold-Status gestellt, gelöscht und anschließend, wenn das END TRANSACTION-Statement ausgeführt wird, aus dem Hold-Status wieder freigegeben.

Anmerkung:

Mit dem Natural-Profilparameter ETEOP kann der Natural-Administrator festlegen, ob Natural am Ende jedes Programms ein END TRANSACTION-Statement generieren soll. Einzelheiten hierzu sagt Ihnen Ihr Natural-Administrator.

Beispiel für ein STORE-Statement:

In dem folgenden Beispiel-Programm werden neue Datensätze der EMPLOYEES-Datei hinzugefügt.

Vorsicht:

Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```

** Example 'STOREX01': STORE (Add new records to EMPLOYEES file)
*
** CAUTION: Executing this example will modify the database records!
*****
DEFINE DATA LOCAL
1 EMPLOYEE-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID(A8)
  2 NAME (A20)
  2 FIRST-NAME (A20)
  2 MIDDLE-I (A1)
  2 SALARY (P9/2)
  2 MAR-STAT (A1)
  2 BIRTH (D)
  2 CITY (A20)
  2 COUNTRY (A3)
*
1 #PERSONNEL-ID (A8)
1 #NAME (A20)
1 #FIRST-NAME (A20)
1 #INITIAL (A1)
1 #MAR-STAT (A1)
1 #SALARY (N9)
1 #BIRTH (A8)
1 #CITY (A20)
1 #COUNTRY (A3)
1 #CONF (A1) INIT <'Y'>
END-DEFINE
*
REPEAT
  INPUT 'ENTER A PERSONNEL ID AND NAME (OR ''END'' TO END)' //
    'PERSONNEL-ID : ' #PERSONNEL-ID //
    'NAME : ' #NAME /
    'FIRST-NAME : ' #FIRST-NAME
  /* *****
  /* validate entered data
  /* *****
  IF #PERSONNEL-ID = 'END' OR #NAME = 'END'
    STOP
  END-IF
  IF #NAME = ' '
    REINPUT WITH TEXT 'ENTER A LAST-NAME'
    MARK 2 AND SOUND ALARM
  END-IF
  IF #FIRST-NAME = ' '
    REINPUT WITH TEXT 'ENTER A FIRST-NAME'
    MARK 3 AND SOUND ALARM
  END-IF
  /* *****
  /* ensure person is not already on file
  /* *****
  FIP2. FIND NUMBER EMPLOYEE-VIEW WITH PERSONNEL-ID = #PERSONNEL-ID
  /*
  IF *NUMBER (FIP2.) > 0
    REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'

```

```

MARK 1 AND SOUND ALARM
END-IF
/*****
/* get further information
*****/
INPUT
'ENTER EMPLOYEE DATA'                ///
'PERSONNEL-ID'           : ' #PERSONNEL-ID (AD=IO) /
'NAME                    : ' #NAME           (AD=IO) /
'FIRST-NAME              : ' #FIRST-NAME    (AD=IO) ///
'INITIAL                 : ' #INITIAL       /
'ANNUAL SALARY           : ' #SALARY        /
'MARITAL STATUS          : ' #MAR-STAT      /
'DATE OF BIRTH (YYYYMMDD) : ' #BIRTH        /
'CITY                    : ' #CITY          /
'COUNTRY (3 CHARS)       : ' #COUNTRY       //
'ADD THIS RECORD (Y/N)   : ' #CONF          (AD=M)
/*****
/* ENSURE REQUIRED FIELDS CONTAIN VALID DATA
*****/
IF #SALARY < 10000
  REINPUT TEXT 'ENTER A PROPER ANNUAL SALARY' MARK 2
END-IF
IF NOT (#MAR-STAT = 'S' OR = 'M' OR = 'D' OR = 'W')
  REINPUT TEXT 'ENTER VALID MARITAL STATUS S=SINGLE ' -
              'M=MARRIED D=DIVORCED W=WIDOWED' MARK 3
END-IF
IF NOT(#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
  REINPUT TEXT 'ENTER CORRECT DATE' MARK 4
END-IF
IF #CITY = ' '
  REINPUT TEXT 'ENTER A CITY NAME' MARK 5
END-IF
IF #COUNTRY = ' '
  REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 6
END-IF
IF NOT (#CONF = 'N' OR= 'Y')
  REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 7
END-IF
IF #CONF = 'N'
  ESCAPE TOP
END-IF
/*****
/* add the record with STORE
*****/
MOVE #PERSONNEL-ID TO EMPLOYEE-VIEW.PERSONNEL-ID
MOVE #NAME         TO EMPLOYEE-VIEW.NAME
MOVE #FIRST-NAME   TO EMPLOYEE-VIEW.FIRST-NAME
MOVE #INITIAL      TO EMPLOYEE-VIEW.MIDDLE-I
MOVE #SALARY       TO EMPLOYEE-VIEW.SALARY (1)
MOVE #MAR-STAT     TO EMPLOYEE-VIEW.MAR-STAT
MOVE EDITED #BIRTH TO EMPLOYEE-VIEW.BIRTH (EM=YYYYMMDD)
MOVE #CITY         TO EMPLOYEE-VIEW.CITY
MOVE #COUNTRY      TO EMPLOYEE-VIEW.COUNTRY
/*
STP3. STORE RECORD IN FILE EMPLOYEE-VIEW
/*
/*****
/* mark end of logical transaction
*****/

```

```

END OF TRANSACTION
RESET INITIAL #CONF
END-REPEAT
END

```

Ausgabe des Programms STOREX01:

```

ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL ID :

NAME          :
FIRST NAME    :

```

Datensatz-Kontrolle während einer Transaktion (Hold-Logik)

Wird Natural zusammen mit Adabas eingesetzt, so wird jeder Datensatz, der verändert werden soll, solange in den Hold-Status gestellt, bis die Transaktion entweder durch ein `END TRANSACTION-` oder `BACKOUT TRANSACTION-`Statement beendet oder aufgrund einer Zeitüberschreitung abgebrochen wird.

Solange ein Datensatz für einen Benutzer im Hold-Status steht, haben andere Benutzer keine Möglichkeit, diesen Datensatz zu ändern. Ein Benutzer, der dies tun will, gelangt in den Wartestatus (Wait) und erhält die Kontrolle über den gewünschten Satz erst, wenn der erste Benutzer seine Transaktion beendet/abgebrochen hat.

Um zu verhindern, dass ein Benutzer im Wartestatus verbleibt, ist es möglich den Session-Parameter `WH` (Wait Hold) entsprechend zu setzen (siehe *Parameter-Referenz-Dokumentation*).

Beim Programmieren sollten Sie folgendes bezüglich der Hold-Logik bedenken:

- Die Zeit, für die ein Datensatz höchstens in den Hold-Status gestellt werden kann, wird von Adabas durch das *Transaction Time Limit* (Transaktionszeitbegrenzung; Adabas `TT`-Parameter) begrenzt. Wird diese Zeit überschritten, erhält man eine entsprechende Fehlermeldung, und Veränderungen, die nach dem letzten `END TRANSACTION-`Statement erfolgten, werden rückgängig gemacht.
- Die Anzahl der ISNs im Hold-Status und mögliche Transaktionszeitüberschreitungen ergeben sich aus der Größe einer Transaktion, d.h. aus der Platzierung des `END TRANSACTION-`Statements. In diesem Zusammenhang sollten Sie die Nutzung von Restart-Möglichkeiten in Betracht ziehen. Falls die Mehrzahl der zu verarbeitenden Datensätze *nicht* verändert werden soll, empfiehlt es sich beispielsweise, ein `GET`-Statement zu verwenden, um das "Halten" von Sätzen zu steuern. Damit spart man viele `END TRANSACTION-`Statements und verringert gleichzeitig die Zahl der in den Hold-Status gestellten ISNs. Bei Verarbeitung umfangreicher Dateien sollte bedacht werden, dass für ein `GET`-Statement ein zusätzlicher Adabas-Aufruf erforderlich ist. Ein Beispiel für die Verwendung eines `GET`-Statements sehen Sie im folgenden.

Beispiel für Hold-Logik:

Vorsicht:

Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```

** Example 'GETX01': GET (put single record in hold with UPDATE stmt)
**
** CAUTION: Executing this example will modify the database records!
*****
DEFINE DATA LOCAL

```

```

1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
END-DEFINE
*
RD. READ EMPLOY-VIEW BY NAME
  DISPLAY EMPLOY-VIEW
  IF SALARY (1) > 1500000
    /*
    GE. GET EMPLOY-VIEW *ISN (RD.)
    /*
    WRITE '=' (50) 'RECORD IN HOLD:' *ISN(RD.)
    COMPUTE SALARY (1) = SALARY (1) * 1.15
    UPDATE (GE.)
    END TRANSACTION
  END-IF
END-READ
END

```

Transaktion abbrechen

Innerhalb einer aktiven logischen Transaktion, d.h. bevor das END TRANSACTION-Statement ausgeführt wird, können Sie durch Verwendung eines BACKOUT TRANSACTION-Statements den Abbruch der Transaktion bewirken. Dadurch werden alle vorgenommenen Änderungen (einschließlich hinzugefügter und gelöschter Datensätze) rückgängig gemacht und die von der Transaktion gehaltenen Datensätze freigegeben.

Transaktion neu starten

Mit dem END TRANSACTION-Statement können Sie auch transaktionsbezogene Informationen speichern. Falls die Verarbeitung der Transaktion nicht ordnungsgemäß beendet werden kann, können Sie beim Neustarten (Restart) der Transaktion diese Informationen mit einem GET TRANSACTION DATA-Statement lesen, um festzustellen, an welchem Punkt die Verarbeitung fortgesetzt werden muss.

Beispiel für Verwendung von Transaktionsdaten beim Neustarten einer Transaktion

Im folgenden Beispielprogramm werden Daten der Dateien EMPLOYEES und VEHICLES verändert. Wenn das Programm nach einem Abbruch neu gestartet wird, werden Sie durch eine Restart-Prozedur darüber informiert, welcher Datensatz der Datei EMPLOYEES vor dem Abbruch zuletzt verarbeitet wurde, und können die Verarbeitung dann an dieser Stelle wiederaufnehmen. Es bestünde zusätzlich die Möglichkeit, Angaben über den zuletzt bearbeiteten Satz der VEHICLES-Datei in die Restart-Transaktionsmeldung einzufügen.

Vorsicht:

Wenn Sie dieses Beispielprogramm ausführen, verändern Sie Datensätze in der Datenbank.

```

** Example 'GETTRX01': GET TRANSACTION
*
** CAUTION: Executing this example will modify the database records!
*****
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'
  ESCAPE BOTTOM
END-NOREC
IF *COUNTER (FIND2.) = 1 /* first pass through the loop
  INPUT (AD=M)
  / 20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
  / 20T '-----'
  /// 20T 'NUMBER:' PERSONNEL-ID (AD=0)
  / 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 IP=OFF) // 28T MAKE / 28T 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

```

Datensätze mit ACCEPT/REJECT auswählen

Dieser Abschnitt behandelt die Statements ACCEPT und REJECT, die Sie zur Auswahl von Datensätzen anhand von Ihnen definierter logischer Auswahlkriterien verwenden können.

Folgende Themen werden behandelt:

- Mit ACCEPT und REJECT verwendbare Statements
- Beispiel für ein ACCEPT-Statement
- Logische Bedingungen in ACCEPT/REJECT-Statements
- Beispiel für ACCEPT-Statement mit AND-Operator
- Beispiel für REJECT-Statement mit OR-Operator
- Weitere Beispiele für ACCEPT- und REJECT-Statements

Mit ACCEPT und REJECT verwendbare Statements

Sie können ACCEPT und REJECT zusammen mit den folgenden Datenbankzugriffs-Statements verwenden:

- READ
- FIND
- HISTOGRAM

Beispiel für ein ACCEPT-Statement

```

** Example 'ACCEPX01': ACCEPT IF
*****
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

```

Ausgabe des Programms ACCEPX01:

Page 1 04-11-11 11:11:11

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

Logische Bedingungen in ACCEPT/REJECT-Statements

Mit einem ACCEPT- oder REJECT-Statement können Sie zusätzlich zu der WHERE- und WITH-Bedingung eines READ-Statements weitere logische Auswahlkriterien angeben.

Das ACCEPT- bzw. REJECT-Auswahlkriterium wird erst ausgewertet, *nachdem* die über das READ-Statement ausgewählten Datensätze gelesen worden sind.

Die folgenden logischen Operatoren können in einem ACCEPT- bzw. REJECT-Statement verwendet werden (weitere Einzelheiten siehe *Logische Bedingungen*):

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

Außerdem können Sie die Boole'schen Operatoren AND, OR und NOT zur Verknüpfung logischer Bedingungen in ACCEPT / REJECT-Statements einsetzen; mit Klammern können Sie die Bedingungen in logische Einheiten unterteilen, siehe folgende Beispiele.

Beispiel für ACCEPT-Statement mit AND-Operator

Das folgende Programm zeigt die Verwendung des Boole'schen Operators AND in einem ACCEPT-Statement:

```
** Example 'ACCEPX02': ACCEPT IF ... AND ...
*****
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
```

Ausgabe des Programms ACCEPX02:

Page 1 04-12-14 12:22:01

NAME	CURRENT POSITION	ANNUAL SALARY
AFANASSIEV	DBA	42800
ATHERTON	ANALYST	43000
ATHERTON	MANAGER	40000

Beispiel für REJECT-Statement mit OR-Operator

Das folgende Programm zeigt die Verwendung des Boole'schen Operators OR in einem REJECT-Statement. Das Programm erzeugt die gleiche Ausgabe wie das vorherige mit dem ACCEPT-Statement, da gleichzeitig die logischen Operatoren umgekehrt wurden:

```
** Example 'ACCEPX03': REJECT IF ... OR ...
*****
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
```

Ausgabe des Programms ACCEPX03:

Page 1

04-12-14 12:26:27

NAME	CURRENT POSITION	ANNUAL SALARY
AFANASSIEV	DBA	42800
ATHERTON	ANALYST	43000
ATHERTON	MANAGER	40000

Weitere Beispiele für ACCEPT- und REJECT-Statements

Siehe die folgenden Beispiel-Programme:

- ACCEPX04 - ACCEPT IF ... LESS THAN ...
- ACCEPX05 - ACCEPT IF ... AND ...
- ACCEPX06 - REJECT IF ... OR ...

AT START/END OF DATA-Statements

Dieser Abschnitt erörtert die Verwendung der Statements AT START OF DATA und AT END OF DATA.

Folgende Themen werden behandelt:

- AT START OF DATA-Statement
- AT END OF DATA-Statement
- Beispiel für AT START OF DATA- und AT END OF DATA-Statement
- Weitere Beispiele für AT START OF DATA- und AT END OF DATA-Statement

AT START OF DATA-Statement

Mit dem Statement AT START OF DATA können Sie eine beliebige Verarbeitung angeben, die ausgeführt werden soll, nachdem der erste Datensatz einer Datenbank-Verarbeitungsschleife gelesen worden ist.

Das AT START OF DATA-Statement muss innerhalb der betreffenden Verarbeitungsschleife stehen.

Erzeugt das AT START OF DATA-Statement eine Ausgabe, so wird diese *vor der ersten Feldwert-Ausgabe* ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

AT END OF DATA-Statement

Mit dem Statement AT END OF DATA können Sie eine beliebige Verarbeitung angeben, die ausgeführt werden soll, nachdem alle Datensätze in einer Datenbank-Verarbeitungsschleife verarbeitet worden sind.

Das AT END OF DATA-Statement muss innerhalb der betreffenden Verarbeitungsschleife stehen.

Erzeugt das AT END OF DATA-Statement eine Ausgabe, so wird diese *nach der letzten Feldwert-Ausgabe* ausgegeben. Standardmäßig erfolgt die Ausgabe linksbündig auf der Seite.

Beispiel für AT START OF DATA- und AT END OF DATA-Statement

Das folgende Beispielprogramm veranschaulicht die Verwendung der Statements AT START OF DATA und AT END OF DATA.

Das AT START OF DATA-Statement enthält die Systemvariable *TIME zur Anzeige der Uhrzeit

Das AT END OF DATA-Statement enthält die Systemfunktion OLD, um den Namen der zuletzt ausgewählten Person anzuzeigen.

```
** Example 'ATSTAX01': AT START OF DATA
*****
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
```

Ausgabe des Programms ATSTAX01:

```
XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT

NAME                CURRENT          INCOME
                   POSITION
                   CURRENCY  ANNUAL    BONUS
                   CODE      SALARY
-----
```

RUN TIME: 12:43:19.1

DUYVERMAN	PROGRAMMER	USD	34000	0
PRATT	SALES PERSON	USD	38000	9000

```
MARKUSH          TRAINEE          USD          22000          0
LAST PERSON SELECTED: MARKUSH
AVERAGE SALARY:          31333
```

Weitere Beispiele für AT START OF DATA- und AT END OF DATA-Statement

Siehe die folgenden Beispiel-Programme.

- *ATENDX01 - AT END OF DATA*
- *ATSTAX02 - AT START OF DATA*
- *WRITEX09 - WRITE-Statement (in Kombination mit AT END OF DATA)*

Unicode-Daten

Natural ermöglicht es den Benutzern, auf Wide-Character-Fields mit dem Format W in einer Adabas-Datenbank zuzugreifen.

In diesem Abschnitt werden folgende Themen behandelt:

- Datendefinitionsmodul
- Zugriffskonfiguration
- Einschränkungen

Datendefinitionsmodul

Adabas Wide-Character-Fields (W) werden auf Natural-Format U (Unicode) abgebildet.

Die Längen-Definition für ein Natural-Feld vom Format U entspricht der Hälfte der Größe des Adabas-Feldes mit dem Format W. Ein Adabas Wide-Character-Field der Länge 200 wird zum Beispiel auf (U100) in Natural abgebildet.

Zugriffskonfiguration

Natural erhält Daten aus Adabas und sendet Daten zurück an Adabas mittels UTF-16 als gemeinsam benutzte Kodierung.

Diese Kodierung wird mit dem OPRB-Parameter angegeben und an Adabas mit der offenen Anforderung versandt. Sie wird für Wide-Character-Fields benutzt und gilt für die gesamte Adabas Benutzer-Session.

Einschränkungen

Wide-character-Fields (W) variabler Länge werden nicht unterstützt.

Sortierfolgen-Deskriptoren werden nicht unterstützt.

Weitere Informationen zu Adabas und Unicode-Unterstützung entnehmen Sie der spezifischen Adabas Produkt-Dokumentation.