CHAPTER 7

Advanced User Topics for the SAS/ACCESS Interface View Engine for IMS-DL/I

Introduction

This chapter includes some considerations for administering the SAS/ACCESS interface view engine for IMS-DL/I (hereafter, referred to as the *IMS-DL/I engine*). It provides additional technical detail on how the engine interface and engine calls work.

Changing the Database and the Effects on Descriptors

Changes to the IMS-DL/I database can affect descriptor files. You must modify or re-create the descriptors if changes to the IMS-DL/I database invalidate the access or view descriptors. Use the ACCESS procedure to edit the affected access descriptors and view descriptors, or to create new descriptors.

If a view descriptor differs from the access descriptor, you receive a message. Re-create or edit the view descriptor as required. If you do not change your descriptor files, IMS-DL/I may return incorrect data to you. If the changes to the database involve numeric data, the procedure that uses the view descriptor could terminate abnormally. See "UPDATE" on page 119 for information on editing descriptors.

Changing an item name has *no* effect on existing view descriptors. However, before you make other changes to IMS-DL/I databases, consider the guidelines described in the next section.

Changes That Cause Existing View Descriptors to Fail

The following changes to the IMS-DL/I database cause existing view descriptors to fail:

- inserting or deleting segments in the middle of the hierarchy if you are updating the database
- inserting or deleting a level in multiple occurring fields
- changing the attributes of a field
- deleting fields that are referenced in a view descriptor
- inserting a field in the middle of a segment
- adding fields to the end of database segments because longer segments may not be reflected in the RECORD statement's SEGLNG= argument.

Ensuring Data Security

This section describes how to prevent unauthorized access to the data in your IMS-DL/I databases.

IMS-DL/I Security

The SAS System preserves the data security that is provided by IMS-DL/I and the operating system. The Database Administrator (DBA) has control over who has access to an IMS-DL/I database. A user cannot use IMS-DL/I facilities through the ACCESS procedure or the SAS/ACCESS interface view engine unless the PSB specified provides that user with the appropriate IMS-DL/I authority. The PSB determines if a user can access an IMS-DL/I database and, if so, the kind of access the user has to the database (Get, Insert, Replace, Delete, or All).

In addition to controlling access to an IMS-DL/I database, the PSB can also control access to specific segments and fields in the database. To control access to a specific database, the DBA can create several view descriptors that describe the same data in the database, and assign each view descriptor a different PSB. Each PSB should define a different type of access to the database. For example, one PSB would allow a user to insert data in the database and another PSB would allow a user only to read the data

in that same database. This enables the DBA to provide each user with a PSB that defines the type of database access the DBA wants to allow that user. Each segment in a view descriptor must be specified in the PSB referenced in the view.

SAS System Security

To secure data from accidental update or deletion, you can do the following on the SAS System side of the interface:

- Set up all SAS/ACCESS access descriptors yourself, dropping items that contain sensitive data so they cannot be referenced in view descriptors. Give users either read-only or no access to the SAS data library where you store the access descriptors. Read-only access prevents users from editing access descriptors and allows them to see only the items selected for each view descriptor.
- Set the IMSDLUPD= or IMSBPUPD= SAS system options to N to disable all updates from the SAS System for a particular region type.
- Assign SAS system passwords (READ, WRITE, ALTER, or PW) to a view descriptor, access descriptor, PROC SQL view, DATA step view, or data file.

  Using passwords adds an extra measure of security if you use view descriptors that include sensitive or confidential data in a shared environment (that is, where SAS/SHARE software is in use). For more information on assigning passwords, see "SAS System Passwords for SAS/ACCESS Descriptors" on page 96.

Maximizing IMS-DL/I Performance

Among the factors that affect IMS-DL/I performance is the size of the database that is accessed. If the database being accessed is very large, you should evaluate all SAS programs that you want to access the database directly. When evaluating the programs, ask the following questions:

- Does the program need all the items included in the view descriptor?
- Does the view descriptor's WHERE statement retrieve only those records or segments that are needed for subsequent analysis?
- Does your WHERE statement directly generate SSAs so that only a subset of the data are passed to the SAS System for processing? To determine whether a WHERE statement is generating SSAs, set the SAS system option IMSDEBUG=Y or set the number of calls for which you want debugging information.

  For HDAM, avoid non-equality conditions in a WHERE statement. See "Identifying Inefficient SAS WHERE Conditions" on page 125 for more information.
- Can you use the DATA step's MODIFY statement to join view descriptors (where each view represents one path in the database) when conditions for a MODIFY's use apply?
- Are the data going to be used by more than one procedure? If so, consider requiring the data to be extracted and placed in a SAS data file rather than accessed directly by each procedure. (See the VIEWDESC= and OUT= options in "PROC ACCESS Statement Options" on page 95 for information on extracting IMS-DL/I data.)

Understanding the IMS-DL/I Interface

This section describes concepts that are exclusive to the SAS/ACCESS interface to the IMS-DL/I engine. You must understand these concepts in order to successfully use the interface. This section describes the following concepts:

- the flattened file concept
- missing values
- special fields
- BY variables

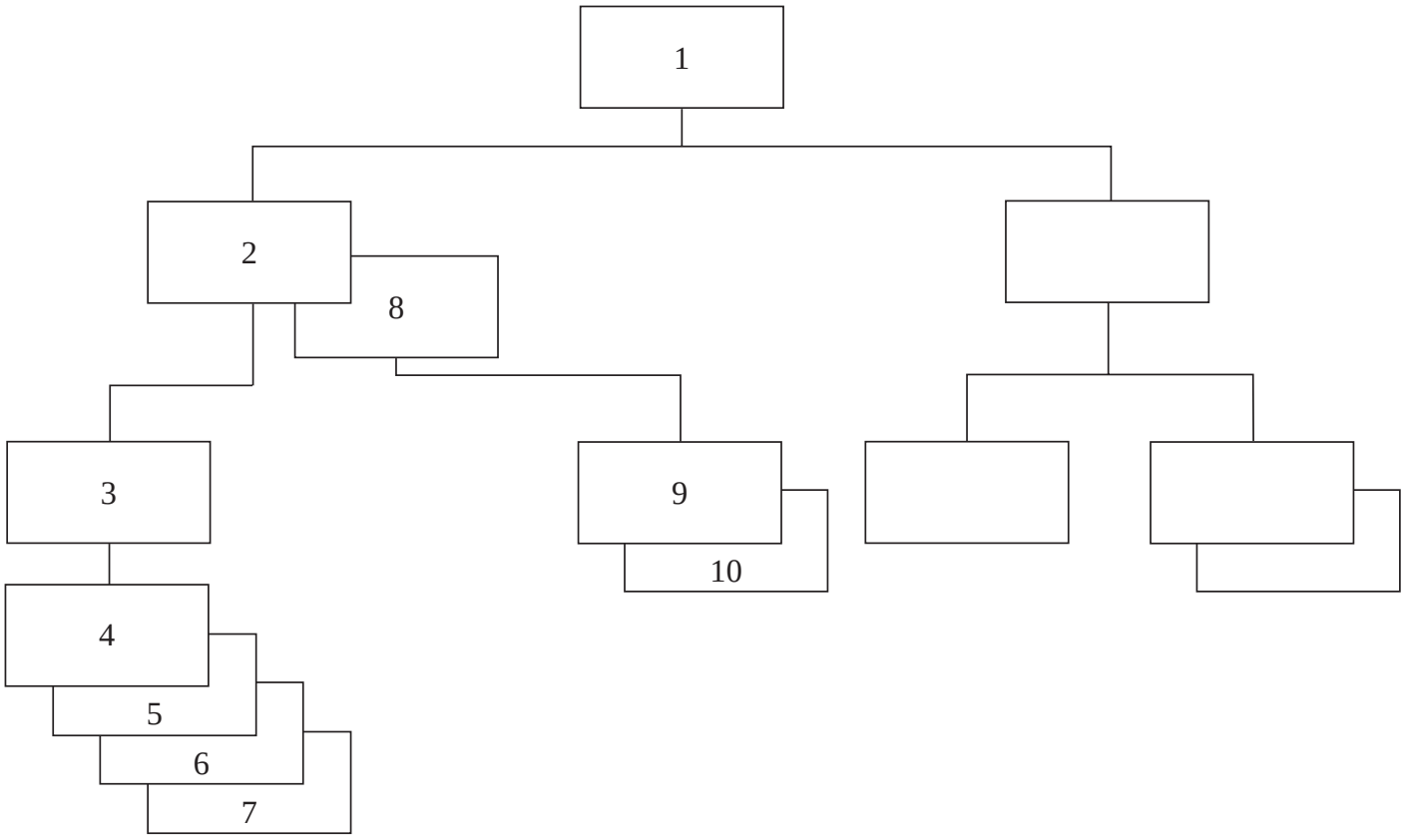
Understanding the Flattened File Concept

When the IMS-DL/I engine creates SAS observations from a hierarchical database, it must flatten out the data. The *flattened file concept* means that the SAS System flattens the hierarchical levels and treats one path of data, including the root segment, parent segments, and child segments, as one SAS observation. If the root segment or any parent segment has children, the parent segment is repeated for each child segment's data. Therefore, each observation contains *all* the parent segments above the child segment.

For example, if you access the data in the database shown in Figure 7.1 on page 133, the IMS-DL/I engine would return data from the following segments as SAS observations. Therefore, the view descriptor would have to define four segment types.

Observation	Segments returned
1	1 2 3 4
2	1 2 3 5
3	1 2 3 6
4	1 2 3 7
5	1 8 9 .
6	1 8 10 .

Figure 7.1 Flattened File Concept



If you use the data from these observations in a SAS procedure, it appears that the data in segment 1 occur six times rather than only once. This can result in misleading statistics when you use such procedures as the MEANS procedure that involves any segment except the child segment in a database with more than two hierarchical levels. It can also be a problem in second-level data because root data repeat. To avoid misleading statistics that can result from flattened files, create view descriptors that describe data in one hierarchical level only. Or perform statistical operations using data only from the lowest level that is accessed by the view descriptor.

Using the *U Command Code

The IMS-DL/I engine generates navigational SSAs to traverse and flatten the database hierarchy. Because sequential calls perform this task, the database's current position is an important issue. (See "Database Position" on page 27 for more information.)

Using a *U command code ensures the current database position on the proper parent segment as a DL/I call moves down the hierarchy to the next target segment (the segment named in the last SSA). *U on the immediate parent of the target means that even if the parent is unqualified, the position indicator remains there and does not move to a child (target segment) that belongs to a different parent occurrence.

For example, when DL/I processes a Get or ISRT call, it establishes a position on the segment occurrence that satisfies the call at each level in the path of the segment (target) that you are retrieving or inserting. * A *U command code on an SSA in a Get or ISRT call tells DL/I not to move from the established database position at the level of the SSA when trying to satisfy the call.

Handling Missing Values

This section describes how the SAS/ACCESS interface to IMS-DL/I handles special data values. It also describes how the DB Content field affects how data are displayed and stored in the database.

If you create a view descriptor to add an IMS-DL/I database segment and fields in that segment are not defined, the IMS-DL/I engine writes low values to the database fields that are not included in the descriptor. The engine does so because it does not know that the fields exist.

If there are missing values in a SAS data set that you use to add or update an IMS-DL/I database, the IMS-DL/I engine writes zeroes to the database for numeric fields and blank spaces to the database for character fields unless you specify a special format (B, L, or H) for the DBCONTENT= argument of the ITEM= statement. DBCONTENT= affects how the engine updates the fields. (See “ITEM=” on page 108 for more information on special formats.)

Conversely, if a field is defined with a DBCONTENT= value and the database retrieves that value (blanks, low values, or high values) in the field, then the IMS-DL/I engine passes missing values to the SAS System. In addition, if a view descriptor describes more than one level in a database, and not all the levels exist for one database record, the IMS-DL/I engine fills the missing segment occurrence with missing values in the SAS observation.

Handling Special Fields

Handling Fields That Occur Multiple Times

An item or a group in an IMS-DL/I database segment can occur more than one time. For example, in the example database ACCTDBD, the two phone number fields, home phone and office phone, could be defined in your access descriptor as one field that occurs two times. To do this, specify OCCURS=2 in the ITEM= statement for the phone number field when you create the access descriptor. When you save the access descriptor, the descriptor is expanded to show fields for two phone numbers. When the IMS-DL/I engine reads the database, it retrieves two phone numbers for each customer.

Fields that occur multiple times in the database can be nested only three levels deep, which creates a three-dimensional table. The following example shows the definition of a record with fields that occur multiple times, nested three levels deep:

```
01 Automatic Teller Record
   02 ATM Information
     03 ATM Location (occurs 20 times)

       04 Location
       04 ATM Transaction Information (occurs 7 times)
```

* You use an ISRT call and I/O or TP PCBs to insert messages to the IMS/ESA control region message queues when a SAS program is executing in a BMP region. See “ISRT Calls to Message Queues” on page 214

```
05 Account Type
05 Transaction Time
05 Transaction (occurs 2 times)

   06 Transaction Type
   06 Transaction Amount
```

After you have saved an access descriptor, you cannot change the number in the OCCURS= argument. Instead, you have to delete an item and re-enter it with the correct number in OCCURS=.

Handling Redefined Fields

Redefined fields are fields that have been defined with more than one data type. For example, some records in a database may have character values stored in a certain field, and other records in the same database may have numeric values stored in that same field. You could handle this by defining the field as **`$11.`** in one access descriptor and **`11.`** in another access descriptor based on the same database. When you create view descriptors for the database, use a WHERE statement to retrieve only the appropriate values for the field. This can often be done by specifying a particular record type or other code in the WHERE statement.

Handling Segments of Varying Length

If you work with a segment that contains a field that varies in length, specify the maximum length of the varying field for SEGLNG= when you define the segment in the access descriptor. When IMS-DL/I retrieves the entire segment, it fills in the varying portion with missing values if it did not retrieve any data for that portion of the segment.

Handling GROUP Keys in Descriptor Files

To support a definition of a GROUP field as a key and to be able to have access to the GROUP items, you need to define a dummy field for this key.

In IMS-DL/I, GROUPs enable the same portion of data in a buffer to be assigned different logical names. For example, a field that begins at offset 1 for a length of 15 can be named FIELD1. Other fields can be defined within FIELD1, such as in FIELD2, FIELD3, and FIELD4 that begin at offsets 1, 6, and 11, respectively (where each has a length of 5).

Because no SAS variable name can be specified in the GROUP= statement, no single reference can be made to the group in the WHERE criteria. Therefore, even if a valid SEARCH/SEQ name exists for the GROUP in the DBD, the IMS-DL/I engine cannot qualify calls that are based on the group itself.

A simple solution is to define the entire group as an item and to assign the SAS variable name and SEARCH name appropriately. Then you can specify a WHERE statement in your view descriptor or application and the IMS-DL/I engine will build qualified SSAs. A problem remains if the application wants access to the components of the GROUP. In this case, you must reference the view descriptor in a DATA step to SUBSTR out the parts and store them in separate SAS variables.

Using Dummy Fields for GROUP Keys

You can define a dummy field in the segment for a GROUP key in order to permit a WHERE clause reference for qualified SSAs and to access the composite fields. The

GROUP statement defines the group but you can take it a step further. You add a dummy field to the end of the segment definition as an ITEM with a length that is equal to the entire GROUP and a SEARCH= value equal to the DBD SEARCH/SEQ field name from the DBD (the GROUP SEARCH= also has this value). The SEGLNG value is increased to allow for this field.

By using a dummy field for the GROUP, you can specify in your view descriptor a WHERE clause as follows:

```
WHERE sas-dummy-name EQ value
```

In this case, the IMS-DL/I engine locates the dummy field in the view descriptor through the SAS variable name in the WHERE clause. It uses its SEARCH= value to qualify the SSA. When the data comes back to the buffer, the true data is in the GROUP portion of the segment definition and its component values are stored in the SAS variables that are associated with the items that are defined for the GROUP.

Also, by marking the GROUP itself as the key (with the KEY= argument), navigational SSAs that are generated by the IMS-DL/I engine for sequential GN calls will refer to the correct buffer location for data. The navigational SSAs will use the correct SEARCH= value in the SSA.

CAUTION:

You must never refer to the dummy field as the key (with KEY=) because doing so would force the IMS-DL/I engine to use the dummy buffer location to qualify navigational SSAs for GN calls. This would cause problems. △

Below is an example of an access descriptor and a view descriptor based on the ACCTDBD. The GROUP key is on home phone, which has a dummy field (GROUP STUFF) defined for it.

```
proc access dbms=ims;
  create work.account.access;
    dbd=acctdbd dbtype=hdam;
    record='customer_record' sg=customer sl=225;
      item=soc_sec_number   lv=2 dbf=$11. key=u
                            se=ssnumber;
      item=customer_name    lv=2   dbf=$40.
                            se=custname;
      item=addr_line_1      lv=2   dbf=$30.
                            se=custadd1;
      item=addr_line_2      lv=2   dbf=$30.
                            se=custadd2;
      item=city             lv=2   dbf=$28.
                            se=custcity;
      item=state            lv=2   dbf=$2.
                            se=custstat;
      item=country          lv=2   dbf=$20.
                            se=custland;
      item=zip_code         lv=2   dbf=$10.
                            se=custzip;
     group=home_phone       lv=2
                            se=custhphn;
      item='area code'      lv=3   dbf=$3.
      item=filler1          lv=3   dbf=$1.
      item=phone_number     lv=3   dbf=$8.
      item=office_phone     lv=2   dbf=$12.
                            se=custophn;
      item='group stuff'    lv=2   dbf=$12.
```

```
                                se=custhphn;
  list all;

  create work.phone.view psbname=acctsam pcbindex=2;
    select soc_sec_number customer_name 'area code'
       'phone number' 'group stuff';
  list view;
run;

proc print data=work.phone;
  var soc_sec_number customer_name 'area code'
      'phone number';
  where 'group stuff' = '803-657-1346' or
        'group stuff' = '803-657-1687';
run;
```

Output 7.1 on page 137 shows the output.

Output 7.1 Results of a Dummy Field for a GROUP Key

The SAS System

OBS	soc_sec_number	customer_name	'area code'	'phone number'
1	436-42-6394	BOOKER, APRIL M.	803	657-1346
2	178-42-6534	PATTILLO, RODRIGUES	803	657-1346
3	434-62-1234	SUMMERS, MARY T.	803	657-1687

Using Filler Notation in ITEM=

It is important that access descriptor segment definitions not omit ITEM and GROUP references for fields that are embedded in the segment. Database segments may contain fields (contiguous or discontiguous) that applications may not need to access. In these cases, it is correct not to define them in SAS/ACCESS view descriptors. For performance reasons, it is recommended that applications not define them so that the IMS-DL/I engine does not invoke conversion routines to convert data that will not be used.

Sites commonly refer to undesired portions of the data buffer by using the FILLER notation in the ITEM= statement, and by defining the DBC (DB Content) as $CHAR. If the undesired portion of the segment lays beyond all the desired segment fields, applications do not have to define these portions of the segment. However, you must be sure that the SEGLNG value for the segment is equal to the length of the entire segment and not just to the portion of the segment that they are interested in defining.

When the undesired fields are embedded between desired fields, you must use the FILLER notation or something similar (FILLER is a reserved word in COBOL but not in SAS). The SAS System uses relative offsets to locate defined fields in the buffer when converting data from the IMS-DL/I buffer to the SAS program data vector (PDV). By using the field lengths from the DBC, SAS determines the offset and length in the IMS-DL/I buffer for the current field as needed to map to the PDV. If a field or series of fields is undesired, information must be supplied about placement and length so that SAS can move correctly to the next valid field to be mapped.

FILLER fields can be coded as DBC of $CHAR, which requires no conversion if selected for a view descriptor. In most cases FILLER fields are not selected. By preserving the relative offsets of fields within the buffer using FILLER definitions, the

IMS-DL/I engine can correctly map data requested by the application or view descriptor to the PDV.

Below is an example of a root segment for the ACCOUNT database with all of the fields defined from the DBD.

```
record='customer_record'   segment=customer
                           seglng=225;
  item=soc_sec_number      lv=2  dbf=$11.
                           search=ssnumber key=y;
  item=customer_name       lv=2   dbf=$40.
                           search=custname;
  item='address info'      lv=2;
  item=addr_line_1         lv=3   dbf=$30.;
  item=addr_line_2         lv=3   dbf=$30.;
  item=city                lv=3   dbf=$28.;
  item=state               lv=3   dbf=$2. ;
  item=country             lv=3   dbf=$20.;
  item=zip_code            lv=3   dbf=$10.;
  item=home_phone          lv=2   dbf=$12.;
  item=office_phone      lv=2   dbf=$12.;
```

Assuming that none of your view descriptors would ever require phone information, you could code the following:

```
record='customer_record'   segment=customer
                           seglng=225;
  item=soc_sec_number      lv=2  dbf=$11.
                           search=ssnumber key=y;
  item=customer_name       lv=2   dbf=$40.
                           search=custname;
  item='address info'      lv=2;
  item=addr_line_1         lv=3   dbf=$30.;
  item=addr_line_2         lv=3   dbf=$30.;
  item=city                lv=3   dbf=$28.;
  item=state               lv=3   dbf=$2. ;
  item=country             lv=3   dbf=$20.;
  item=zip_code            lv=3   dbf=$10.;
```

Note that the SEGLNG= value does not change even though two fields at the end are dropped.

By comparison, assume that the application needs everything except the address information:

```
record='customer_record'   segment=customer
                           seglng=225;
  item=soc_sec_number      lv=2   dbf=$11
                         .  search=ssnumber key=y;
  item=customer_name       lv=2   dbf=$40.
                           search=custname;
  item='filler'            lv=2   dbf=$char60.;
  item=city                lv=3   dbf=$28.;
  item=state               lv=3   dbf=$2. ;
  item=country             lv=3   dbf=$20.;
  item=zip_code            lv=3   dbf=$10.;
  item=home_phone          lv=2   dbf=$12.;
  item=office_phone        lv=2   dbf=$12.;
```

Here, the FILLER preserves 60 bytes so that view descriptors that reference fields past the filler can get data mapped correctly from the IMS-DL/I buffer to the PDV variables based on the relative offset information. Once again, SEGLNG= does not change.

Using BY Variables

If you specify an IMS-DL/I view descriptor as input to a SAS procedure that uses a BY variable, you must either

- create a SAS data file from the IMS-DL/I data (that is, extract the data) and sort the data using that variable. You then specify the newly created data file in your procedure.
- reference a SAS variable associated with a database index in the BY statement; that is, the BY variable must be defined as the index key.

Understanding the IMS-DL/I Engine Calls to the Database

To create an access descriptor using the ACCESS procedure, you must first enter the database definition. IMS-DL/I does not store descriptive information about databases in a dictionary or database. After you have created an access descriptor, you can select variables from one path of data when you create a view descriptor. The IMS-DL/I engine is designed to get its information to build its own SSAs from the view descriptors and any supplied WHERE clause; these views are based on access descriptors that define the DL/I databases. The IMS-DL/I engine uses the information stored in the view descriptor to generate IMS-DL/I calls and to format the results of those calls into SAS observations. By design, view descriptors cannot access IMS/ESA control region message queues. Therefore, the IMS-DL/I engine interface is not able to access the message queues if it is executing in a BMP region.

Data Retrieval

The IMS-DL/I engine sequentially processes database data in order to flatten IMS records when no WHERE criteria exist. All data in the path specified by the view descriptor are returned in the order in which they were stored in the database when you use unqualified Get-Next (GN) processing. Therefore, the IMS-DL/I engine uses qualified segment search arguments (SSAs) to navigate the database path and maintain proper positioning, basing all qualified Get calls on the results of the previous call. You can use SAS WHERE statements to perform some level of direct access to a database.

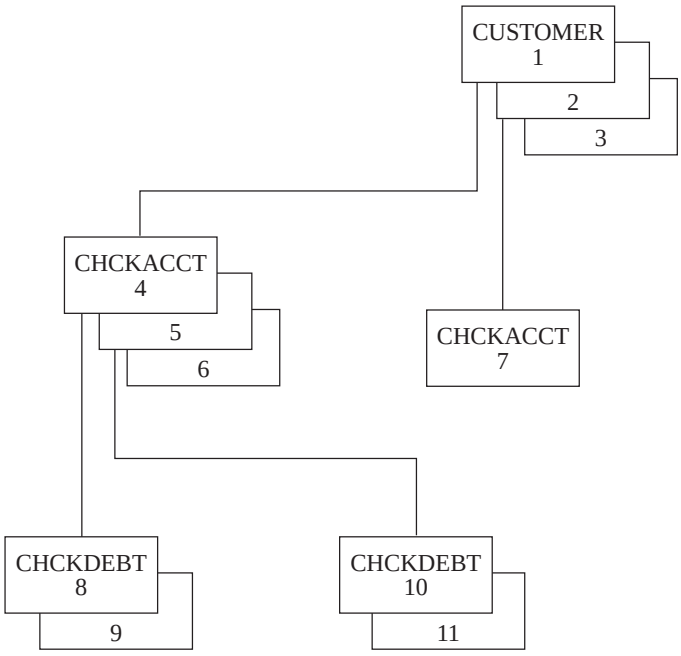
You can see an example of this process by using the view descriptor VLIB.CHKDEB, which describes the CUSTOMER, CHCKACCT, and CHCKDEBT segments in the ACCTDBD database. First, the IMS-DL/I engine issues an unqualified Get Unique (GU) call to position itself at the beginning of the database. If the CUSTOMER segment were the only segment in the ACCTDBD database, the IMS-DL/I engine would then issue qualified Get Next (GN) calls for CUSTOMER until it reached the end of the database. However, because the ACCTDBD database is a multilevel database and the view descriptor defines more than the root segment, the processing is more difficult. To obtain the dependent segment, the IMS-DL/I engine must use the value returned in the I/O area for the field designated as the key in order to build a qualified SSA for the parent segment (in this case the root segment).

Next, the IMS-DL/I engine issues a Get Next Within Parent (GNP) call by concatenating the qualified SSA for the root segment with an unqualified SSA for the

next level down in the hierarchy. The engine then takes the value of the field designated as the key field of that segment (as defined originally in the access descriptor) from the I/O area to generate a qualified SSA for that level. The next database call is a GNP with the two qualified SSAs concatenated with an unqualified SSA for the next level down in the hierarchy. The engine continues to combine qualified SSAs with an unqualified SSA for the next lowest level down the hierarchy until the lowest level (as defined in the view descriptor) is retrieved, or until a status of GE is returned; GE indicates no segment occurrence.

Figure 7.2 on page 140 shows the segments described by the view descriptor, VLIB.CHKDEB, and the order in which the segments are accessed by IMS-DL/I. The calls generated by the IMS-DL/I engine to navigate the database are also described. Note that one SAS observation is made up of one complete path of data. If there is no child segment, the IMS-DL/I engine passes missing values in the fields for that segment to the SAS System.

Figure 7.2 ACCTDBD Segments Described by VLIB.CHKDEB



Shown below is the call output that is generated by the IMS-DL/I engine when it navigates the database (based on Figure 7.2 on page 140). It is printed to the SAS log by using the SAS system IMSDEBUG=Y. It shows how the IMS-DL/I engine uses the *U command code to maintain parentage in cases where no key field has been defined for one or more hierarchical levels in the view descriptor. See “Using the *U Command Code” on page 133 for more information.

```
GU                                    gets CUSTOMER 1
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)  gets CHCKACCT 4
CHCKACCT*--
Status Code=

GNP
```

```
CUSTOMER*U-(SSNUMBEREQ667-73-8275)  gets CHCKDEBT 8
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKDEBT 9
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)  gets CHCKACCT 5
CHCKACCT*--
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKDEBT 10
CHCKACCT*U-(ACNUMBEREQ345620154633)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKDEBT 11
CHCKACCT*U-(ACNUMBEREQ345620154633)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKACCT 6
CHCKACCT*--
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*U-(ACNUMBEREQ345620180723)
```

```
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*--

Status Code=GE

GN
CUSTOMER*--                          gets CUSTOMER 2
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ434-62-1234)  gets CHCKACCT 7
CHCKACCT*--
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ434-62-1234)
CHCKACCT*U-(ACNUMBEREQ345620104732)
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ434-62-1234)
CHCKACCT*--

Status Code=GE
GN
CUSTOMER*--                         gets CUSTOMER 3
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ436-42-6394)
CHCKACCT*--

Status Code=GE

GN
CUSTOMER*--

Status Code=GB
```

Note: The data retrieval process for GSAM databases is somewhat different. After issuing an initial close call (CLSE) to establish position at the beginning of the database, the IMS-DL/I engine uses unqualified GN calls to retrieve all the data in the database. △

WHERE Statement Processing

There are many ways to subset data in the SAS System by using

- a WHERE statement in a view descriptor
- a SAS WHERE statement in a PROC or DATA step
- a PROC SQL SELECT statement's WHERE clause
- a WHERE command in the SAS/FSP procedures
- a SAS data set WHERE option.

These all use SAS WHERE statement syntax. You do not have to use IMS-DL/I SSA syntax with the IMS-DL/I engine that runs under Version 7 of the SAS System.

The IMS-DL/I engine attempts to build SSAs from the WHERE conditions that you enter; *condition* refers to the expression(s) in the WHERE statement, clause, command, or option. The engine uses these SSAs to qualify each call to the database. Therefore, IMS-DL/I returns to the SAS System only those observations that meet your conditions. However, if the IMS-DL/I engine cannot convert the WHERE condition into SSAs, it passes all database segments referenced by the view descriptor to the SAS System, which then subsets and processes the data. Because it uses more resources to have the SAS System process WHERE conditions, you should try to use WHERE conditions that can be turned into valid SSAs when resources are a concern.

To specify WHERE conditions that the IMS-DL/I engine can use to generate SSAs, use one of the operators supported by IMS-DL/I. In the access descriptor, define search field names from the DBD for all the variables included in your WHERE condition when possible. See "Writing Efficient WHERE Statements" on page 124 for a list of the operators IMS-DL/I supports.

Note: IMS-DL/I SSAs do not support OR'd conditions from two different segments. △

The engine uses the search field names that are entered in the view descriptor for the field names in the SSAs. Therefore, if you use a SAS variable in a WHERE condition for which you do not have a search field defined, the IMS-DL/I engine cannot generate SSAs for that WHERE condition.

If the WHERE statement or clause contains multiple conditions and any one of the conditions cannot generate a qualified SSA, then no qualified SSA is generated from the statement or clause.

If the IMS-DL/I engine can handle a WHERE condition, it uses the SEARCH= argument in the ITEM= statement to generate a qualified SSA. If possible, the engine combines the qualified SSAs that it generated to navigate the database with any WHERE condition SSAs. If both SSAs involve the same field, only the WHERE SSA is used to avoid a mutually exclusive situation. The engine then issues a path call to obtain the segments in the hierarchy down to the lowest level with an item specified in the WHERE condition. All segments in the path are retrieved and passed to the SAS System. Therefore, if you use a WHERE condition from which the IMS-DL/I engine can generate SSAs, the Program Specification Block (PSB) specified in that view descriptor must allow path calls for the segments in the hierarchy above and including segments with variables in the WHERE condition.

For example, if you enter the WHERE condition

```
WHERE CHCKACCT = 345620145345
```

the IMS-DL/I engine passes the following SSAs to IMS-DL/I:

```
CUSTOMER*D-
CHCKACCT*--(ACNUMBERREQ345620145345)
```

The IMS-DL/I engine uses the results of this call to generate SSAs to navigate the database further and to flatten out the IMS record as defined in the view descriptor. The engine combines these navigational SSAs with the SSA that it generated from the WHERE condition for the CHCKACCT segment. The engine continues processing and retrieves the view descriptor's lowest level segment (CHCKDEBT), which is a child of

the CHCKACCT segment. CHCKACCT has an ACNUMBER value that is equal to 345620145345 until the engine does not find another CHCKDEBT segment (status code GE).

To improve the efficiency of using a WHERE condition to subset your data, use the operators supported by IMS-DL/I. Enter the search field names of all variables in the WHERE condition so that the IMS-DL/I engine can pass only a subset of data to the SAS System for further processing. Use the SAS system option IMSDEBUG=Y to see whether your WHERE condition is generating SSAs directly.

Note: For GSAM databases, the IMS-DL/I engine always passes WHERE clauses to the SAS System for processing. △

Data Retrieval by Using a Secondary Index

The SAS/ACCESS interface enables you to take advantage of secondary indexes in IMS-DL/I. A secondary index enables a SAS application to:

- access a segment type in the database in a sequence other than the sequence that is specified by the key field. For example, the application might need to access a database by phone numbers—a field in the root segment of the database—rather than by the social security number, which is the segment's key.
- change the view of the database data based on a condition in a dependent segment in the database. For example, a banking application might need to access the database (normally accessed by the social security number, the root segment's key field) by the checking account number or the savings account number, which are key fields in dependent segments to the root segment.

Because IMS-DL/I stores root segments in the sequence of their key fields, an application that accesses the data in another order would be inefficient. A database administrator (DBA), in conjunction with the SAS application user analyst/programmer, determines if a secondary index is needed and assists in laying out the secondary index. By using secondary indexing, IMS-DL/I can go directly to a segment based on a field that is not the key field.

You can define your access descriptors and view descriptors so that they can access secondary indexes, as described in this section.

In IMS-DL/I, when an application requests that a segment be returned based on the database call and SSA combination, the segment that is returned is called the *target segment*. If an application requests only one segment, that segment becomes the target segment. If a sequence of SSAs is used, then the lowest level segment retrieved in the hierarchy is the target segment. If you issue a path call for multiple segments, all the segments are returned to the I/O area.

To use secondary indexes with SAS applications, you have to assign certain IMS-DL/I parameters and use certain arguments when you create an access descriptor. The PCB that you use must specify the PROCSEQ parameter, which causes IMS-DL/I to use the secondary index. You also may need to use the PCBINDX= argument when you create a view descriptor so that the correct PCB is used by the engine. The secondary index is transparent to the application because it is automatically accessed when these parameters are assigned.

To create a secondary index, the DBD for the database must contain XDFLD statements to

- define the name of an indexed field that is associated with an index target segment type
- identify the index source segment type and
- identify the index source segment fields that are used in creating a secondary index.

One XDFLD statement is required for each secondary index relationship.

Combining Segments to Define Descriptors

This section lists ways in which target and source segments can be related and, therefore, how you should define your descriptors in order to access the IMS-DL/I data through a secondary index.

1. When the source segment and the target segment are the same, and the target segment is the root segment:

   The *source segment* supplies the field(s) values that comprise the secondary index. The secondary index stores these values in order with other information that specifies where the target segment is located for any value of the secondary index.

   The XDFLD statement contains the NAME= value that is used in the SEARCH= argument, because doing so gives the secondary index the same name as will be used in the application's SSAs.

2. When the source segment and the target segment are the same, and the target segment is *not* the root segment:

   If the target segment is not the root segment, the database is conceptually restructured. The DBA and the SAS applications analyst/programmer lay out how the database will look conceptually. Physically, the database is still the same. This causes the SAS application to access the data by using the secondary index data structure of the database. For this case, in addition to the scenario described in (1), the entire access descriptor definition must describe the secondary index data structure and not the primary structure.

3. When the source segment and the target segment are *not* the same, and the target segment is the root segment:

   In this case, there is no secondary index data structure because the target segment is the root segment. However, the target and source segments are separate segments in the database.

   In order for you to create an access descriptor using separate segments, you must add a dummy field to the end of the root segment. This dummy field must contain a length that matches the field(s) length for the target segment key value. In addition, the SEGLNG value for the entire root segment must be increased in the DBD to allow for this additional field. Any valid SAS name can be assigned to this dummy field, but the SEARCH= value must be the XDFLD name for the field from the DBD.

   In essence, the dummy field is a virtual field in the access descriptor definition for the root. It does not physically exist there, but a SAS application can submit SSA references for the target (in this case the root) that is qualified on this field.

   For example, consider a SAS application that uses

   ```
   WHERE sasname EQ value
   ```

   where *sasname* is the SAS variable name for the virtual field and *value* is a value for the field in the source segment. The IMS-DL/I engine properly builds a SSA for the target (root) that is qualified by using the XDFLD name for the field and the value from the WHERE clause.

4. When the source segment and the target segment are *not* the same, and the target segment is *not* the root:

   This is the most complicated. It combines the scenarios described in (2) and (3). The same dummy field must be added to the target segment as in (3). In addition,

the entire access descriptor must map to or define the secondary data structure that results from the target not being the root.

Data Modification Processing

Modifying a hierarchical database such as IMS-DL/I can be complicated. Therefore, you need to know how the IMS-DL/I engine operates in order to perform database modifications.

If you plan to use a view descriptor to update the database, the IMS-DL/I engine requires that you designate one search field as a key (that is, one key field) for each hierarchical level in the database. You designate the key field when you create the access descriptor on which the view descriptor is based. The key fields must be selected in the view descriptor.

Note: The search field that you designate as the key *must* be defined in the DBD as a key field; otherwise, updating results may be unpredictable. In addition, you cannot skip hierarchical levels in a view descriptor that you want to use to update the database. Because the IMS-DL/I engine uses path calls to perform most updates, no ROLB (ROLLBACK) calls are required. If a path call fails, the engine returns an error to the SAS System and no update is performed. △

The engine, by default, issues checkpoints at the beginning and end of the update process. You can use the AUTOSAVE option with SAS/FSP software to increase the frequency of issuing checkpoints. Your update PSB must allow path call processing, and an I/O PCB must be included for checkpoint calls.

The only time an update is performed with multiple IMS-DL/I calls is when you request both an update and an insert. For example, you could use the FSEDIT procedure to update a CUSTOMER segment and, on the same display, enter information to insert a new CHCKACCT segment under the CUSTOMER parent segment you just modified. In this case, if the insert call fails after the engine has processed the modification, the IMS-DL/I engine issues another update call that replaces the modified parent segment with the original data in that segment. This process uses fewer resources than a ROLB call. (See Chapter 9, “Using the SAS/ ACCESS Interface to IMS-DL/I DATA Step Interface,” on page 197 for information on ROLB and other non-database access calls.)

Delete Processing

You can delete only the lowest existing segment defined in the view descriptor.

CAUTION:
If you delete a segment that has children that are not defined in the view descriptor, the children are also deleted. △

For example, if your view includes the CUSTOMER and CHCKACCT segments only and you delete a CHCKACCT segment, any CHCKDEBT segments under CHCKACCT are also deleted even though they are not defined in the view descriptor.

If your view descriptor includes all the hierarchical levels but a particular segment has no children, the lowest existing segment is deleted. For example, if a CUSTOMER segment occurrence has no CHCKDEBT segments under a CHCKACCT segment, issuing the DELETE command deletes the CHCKACCT segment. If you then have only a CUSTOMER segment and you issue the DELETE command, the CUSTOMER segment is deleted.

Note: You cannot delete segments in a GSAM database. △

Add Processing

There are three ways to insert new data in an IMS-DL/I database by using SAS/FSP software:

- To add a path of data to your database, enter the new data using the FSVIEW procedure (with the MODIFY command) or the FSEDIT procedure, and issue the ADD command. The IMS-DL/I engine adds all the data that you entered as a new path of data in your database.
- To add a new child segment under an existing root segment that does not have any children, use PROC FSVIEW (with the MODIFY command) or PROC FSEDIT to display the existing segment. Enter the child data on the screen below the existing parent segment.
- To add a twin segment to an existing child segment, you must first display the segment to which you want to add a twin. Enter the new data by typing over the existing child, making sure you change the key field of the segment to which you want to add a twin. The IMS-DL/I engine then inserts a twin segment. Any segments which appear on the screen under the changed segment are also added under the new twin segment in a path call.

  Note that you can add segments only at the end of a GSAM database.

You can also use the APPEND procedure, DATA step MODIFY statement, or an INSERT statement in the SQL procedure to add data to an IMS-DL/I database. To insert a path of data, use a view descriptor that describes the entire path to be inserted. To insert child segments under a parent segment, enter the key field value of the parent segment. The new data will be inserted under the existing parent.

Update Processing

The IMS-DL/I engine compares the data you entered to the data that are stored in the I/O area from the last call. If you change any data in a path of data, the engine replaces only the segments that have changed in the path. If you change the key field defined in the view descriptor, the IMS-DL/I engine inserts a twin segment occurrence under the current parent segment.

Note: You cannot update segments in a GSAM database. △

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/ACCESS® Interface to IMS-DL/I Software: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999. 316 pp.

SAS/ACCESS® Interface to IMS-DL/I Software: Reference, Version 8

ISBN 1–58025–548–5

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999