Advantage™ VISION:Inform®

Concepts Guide

4.0

Computer Associates®

IFKCC040.PDF/D39-080-011

Contents

Chapter 1: Introducing Advantage VISION:Inform

Chapter 2: Understanding VISION:Inform Concepts

Chapter 3: Understanding VISION:Inform Architecture

Chapter 4: Understanding Files

Chapter 5: Understanding Logical Data Views

Chapter 6: Understanding Tables

Chapter 7: Understanding Procedures

Chapter 8: Processing Data Structures

Chapter 9: GDBI Processing

Index

Chapter

1

Introducing Advantage VISION:Inform

This manual explains the concepts and features of Advantage™VISION®:Inform. It also explains how you can use these concepts and features to get the most out of your VISION:Inform system.

This document is for any VISION:Inform user, the VISION:Inform system administrator, and the VISION:Inform database administrator. When the text is specifically for the individuals who perform the system administration and database administration tasks, the text will use the terms “system administrator” and “database administrator,” respectively.

Chapter 2, *Understanding VISION:Inform Concepts* describes some of the unique concepts and features that make VISION:Inform both a powerful and easy to use data extraction tool.

Chapter 3, *Understanding VISION:Inform Architecture* discusses the architecture of VISION:Inform. The VISION:Inform environment consists of individual component pieces that, when used together, form a complete, comprehensive, and cohesive data extraction environment. You will see how VISION:Inform addresses all aspects of the data extraction process.

Chapter 4, *Understanding Files* provides a detailed discussion of the VISION:Inform file definition. The file definition provides VISION:Inform with a map of your physical data. The data organization, structure, and relationships are all defined in the file definition. You can use the file definition to logically re-map your physical data into new, more useful, logical structures called logical records.

Chapter 5, *Understanding Logical Data Views* provides a detailed discussion of logical data views, the VISION:Inform capability that enables you to join all forms of data bases.

Chapter 6, *Understanding Tables* provides a detailed discussion of VISION:Inform table definitions. Table definitions provide you with an automatic lookup capability. During the data extraction process, terse codes can automatically be replaced with descriptive words and phrases.

Chapter 7, *Understanding Procedures* provides a detailed discussion of procedures. VISION:Inform provides you with a simple, but powerful, free-form language so that you can automatically tie procedural logic to the data extraction process.

Chapter 8, *Processing Data Structures* explains the ways in which data structures are processed by VISION:Inform, both automatically and under controlled conditions.

Chapter 9, *GDBI Processing* provides a detailed discussion of VISION:Inform Generalized Data Base Interface (GDBI). Using standard VISION:Inform data extraction methods, you can retrieve data from any standard IBM® host computer data base manager. However, in addition to processing all standard data base types, VISION:Inform also lets you process any Independent Software Vendor (ISV) data bases by using the GDBI processing option.

Related Documents

VISION:Inform documentation includes the following books:

- *Advantage VISION:Inform Getting Started Guide*

  Contains information pertinent to the current release of the product, and information regarding any special conversion and migration issues for those upgrading from prior releases.

- *Advantage Vision:Inform Release Summary*

  Describes the new features of Advantage VISION:Inform.

- *Advantage VISION:Inform for CICS Installation Guide*

  Describes installing, tailoring, and maintaining the components of VISION:Inform in the CICS environment.

- *Advantage VISION:Inform for IMS/DC and IMS/TM Installation Guide*

  Describes installing, tailoring, and maintaining the components of VISION:Inform in the IMS environment.

- *Advantage VISION:Inform Concepts Guide*

  Explains the different types of definitions used by VISION:Inform, including table, file, and procedure definitions. This book also describes the Promote process, which compiles your definitions into an executable format.

- *Advantage VISION:Inform System Administrator Guide*

  Provides information essential to the system administrator and anyone responsible for supporting VISION:Inform. It explains the controls needed for users to access database information while maintaining security and proper use of system resources.

- *Advantage VISION:Inform Definition Processor Reference Guide*

  Describes defining tables, files, logical data views, and procedures for use by VISION:Inform. It discusses, in detail, the concepts, considerations, and specifications for each definition type. This book also provides detailed information on the use of the Definition Processor, the interactive facility for creating and maintaining your definitions.

- *Advantage VISION:Inform ASL (Advanced Syntax Language) Reference Guide*

  Contains information specific to the use and operation of ASL, the free-form language you use to create procedures for VISION:Inform. The reference includes a full description and example of each procedure statement.

- *Advantage VISION:Inform User Guide*

  Describes the use of VISION:Bridge, the VISION:Inform subsystem host 3270 client, for query processing and reporting. It contains complete information on command statement syntax for VISION:Bridge and Immediate Response functions and commands.

- *Advantage VISION:Inform for CICS Utilities Guide*

  Contains information on the utilities that come with VISION:Inform for CICS.

- *Advantage VISION:Inform for IMS Utilities Guide*

  Contains information on the utilities that come with VISION:Inform for IMS.

  The utilities are supplied for the convenience of the system administrator and include the following:

  - COBOL Quick Start Utility
  - Communication File Backup Utility
  - Communication File Purge Utility
  - Communication File Restore Utility
  - DB2 Quick Start Utility
  - Definition Convert Utility
  - Field Description Merge Utility
  - Glossary Utility
  - Initialize Utility
  - Library Backup Utility
  - Library Restore Utility
  - Promote Process Utility
  - Query Migration Utility
  - VISION:Builder® Quick Start Utility
  - VISION:Inquiry® Quick Start Utility
  - VISION:Results™ Quick Start Utility

- *Advantage VISION:Inform Messages and Codes*

  Contains all of the messages and codes issued by the system, along with an explanation of each.

Contacting Computer Associates

For technical assistance with this product, contact Computer Associates Technical Support on the Internet at esupport.ca.com. Technical support is available 24 hours a day, 7 days a week.

Chapter

2

Understanding VISION:Inform Concepts

This chapter provides an overview of the concepts and powerful capabilities provided by VISION:Inform.

- An overview of VISION:Inform.
- VISION:Inform and the Client/Server Environment — The role of VISION:Inform as a server in a client/server environment.
- Getting the Most Out of VISION:Inform — The nature of the databases and the environment for which VISION:Inform is best suited.
- VISION:Inform and the Virtual Relational Table — The unique capability of treating combinations of different databases (DB2, IMS, VSAM, and others) as if they were from one relational table.
- Automatic Navigation and Set Operation — The capability of returning all desired data, with all relationships preserved, in a single data request.
- The Logical Record — The unique capability of solving problems by defining data structures rather than writing procedural code.
- The Logical Data View — The ability to join different databases (such as DB2, IMS, VSAM) and provide automatic transformation or data manipulation procedures.
- VISION:Inform and the Virtual Relational Table — The object types supported and the component by which they are defined.
- The Definition Library — The repository for all VISION:Inform objects.
- The Definition Process — An overview of the architecture of VISION:Inform.
- Putting it all Together — The interaction of the different concepts.
- A Summary of VISION:Inform features.

VISION:Inform

VISION:Inform is a powerful, intelligent data extraction product that provides all workstation users with access to all data stored on your IBM host computer.

- Intelligent extraction means that VISION:Inform does not just download data; it can automatically synchronize, match, transform, filter, and summarize data as well.
- VISION:Inform leverages both the computational power of the host computer and the user-friendly environment of the workstation, completing the work on the platform that is best suited to the type of processing being performed.

By exploiting modern client/server technology, VISION:Inform provides you with a familiar, user-friendly environment of the workstation, using an English-like language, or a friendly graphical user interface (GUI), to request the exact subset of data that is needed to complete the specific task at hand.

- The user-friendly environment means that you do not need knowledge of the format or location of the actual physical data that is being requested from the server. This is because VISION:Inform makes the mechanics of the actual data extraction process transparent to you without you having to learn and remember a complicated programming language.
- You do not need to concern yourself with database managers, physical storage mediums, or database organization.
- You do not need to learn terms like CICS, DB2, IMS, JCL, OS/390, or VSAM. The only prerequisite is that you are able to identify the data you need. VISION:Inform takes care of the rest.

Using VISION:Inform technology to provide the seamless data extraction capabilities provides you with the means of putting strategic data in the hands of the people that need it, when they need it.

- Basing important business decisions on last week's data is no longer necessary as you can request and receive data when you need it with VISION:Inform.
- VISION:Inform gives you the information you need to make informed decisions and provides you with accurate, up-to-date answers.
- Time consuming requests for data to the Information Systems organization are no longer needed. Data is accessible when you need it, not when a programmer becomes available.
- Searching through large amounts of data, in search of specific information, is also eliminated as you can select an exact subset of data for downloading. The downloading of unnecessary, extraneous data is eliminated. With VISION:Inform, generic data downloads are a thing of the past.

VISION:Inform provides all workstation users easy, seamless access to all your host computer data regardless of the data type, organization, access method, or physical storage medium.

VISION:Inform and the Client/Server Environment

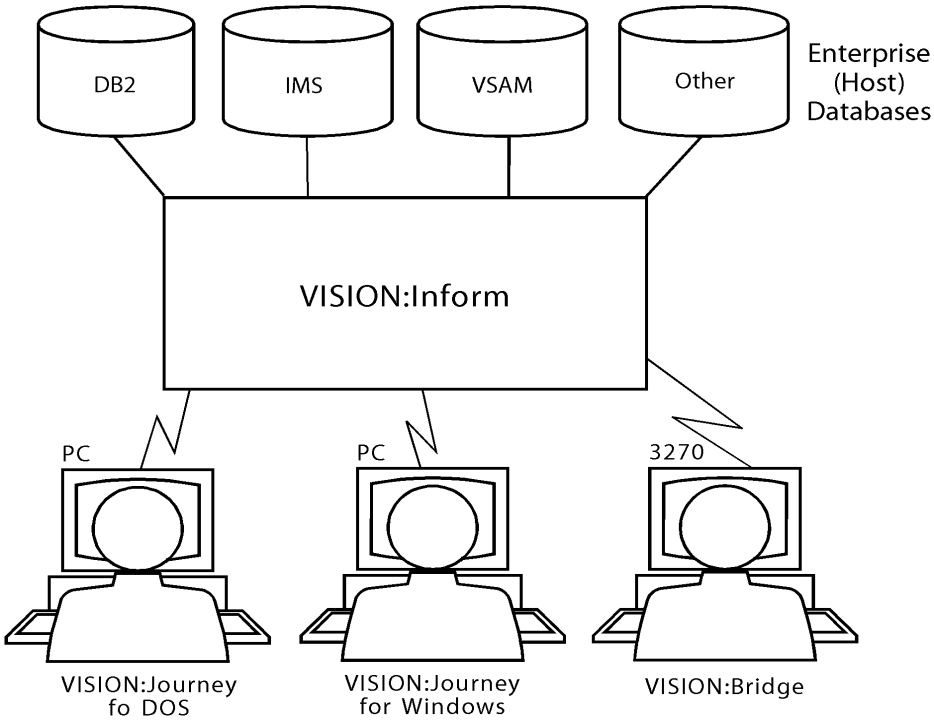

Figure 2-1 The Client/Server Environment

As shown in Figure 2-1, VISION:Inform uses client/server technology to provide access to any type of data stored on IBM host computers from a variety of clients such as:

- VISION:Bridge from the 3270 on the host.
- VISION:Journey for DOS and VISION:Journey for Windows from the workstation.

VISION:Inform gives you a single, centralized server that can simultaneously service multiple clients. All requests for data funnel through one common server providing a central point of control for all data retrieval.

Having a single central server provides the following benefits:

- The data extraction process works the same for all client platforms, providing consistency for all users.
- You maintain only one central server on the host computer.
- You can define and enforce data access security profiles from a central point.
- You can control and optimize system resources from a central point.
- One server can simultaneously handle a large number of users.

VISION:Inform turns your host computer and all its databases (DB2, IMS, VSAM, and others) into a centralized server which can simultaneously service all of your data retrieval needs.

Getting the Most Out of VISION:Inform

You can divide requests for data retrieval into two basic categories:

- Requests for small amounts of data that require immediate or interactive response.

  Example: The customer support operator who needs access to a single customer record in a formatted, interactive environment that provides for browsing of records.
- Requests for large amounts of data that provides for scheduled response.

  Example: The company accountant who needs to perform analysis of all accounts over the last three years. The accountant retrieves data for analysis and reporting purposes only.

The VISION:Inform strong point is the latter category. VISION:Inform is designed for extraction of large amounts of corporate data, from large production databases, for analysis by workstation users.

Frequently, large production databases can only be passed through once in a 24 hour period. You simply cannot afford to make multiple passes of large production databases in a single day. You must optimize processing by satisfying all of your data extraction needs from all of your users simultaneously in one pass of the database.

VISION:Inform lets you do this automatically. It is absolutely essential that you control when the requests for data are processed and which of your many users are to receive priority processing. The VISION:Inform scheduled response capability provides you with the means of controlling the use of resources.

Scheduled response simply means that your VISION:Inform system administrator can define different data extraction response classes. Using these response classes, you control when and how data extraction requests are processed.

- Different response classes can be set up for data extraction requests to be processed immediately, overnight, once a week, or anywhere in between.
- Response classes can also be set up to group similar, but separate, data extraction requests together so they can be processed as one big request. This means that multiple data extraction requests, normally requiring multiple passes of the database, can be processed together in only one pass of the database. Once the data is retrieved, VISION:Inform separates the retrieved data and routes it back to the appropriate users.

Scheduled extraction provides the flexibility to meet the specific needs of your company.

- Scheduled extraction means you can schedule non-critical processing to off hours when the system is otherwise idle, yet you still have the flexibility to meet critical situations where rapid response is required.
- Scheduled extraction gives you the power to control your system resources, more evenly distribute the processing loads placed on your system, and reduce the number of database access requests required to fill your user requests.

Host processing time is a valuable commodity in the business environment of today. VISION:Inform gives you the power to use this resource not only more efficiently, but more effectively as well.

VISION:Inform and the Virtual Relational Table

VISION:Inform provides end users with the ability to retrieve host data without knowing the format or location of the actual physical data being requested. This way, the mechanics of the actual data extraction process are transparent to the users.

Seamless, transparent data extraction is made possible by a powerful concept that is unique to VISION:Inform. This is the concept of the virtual relational table.

VISION:Inform gives your database administrator the ability to present your corporate data as a simple relational table to end users. We call this relational view of your data a virtual relational table, because it does not really exist in physical storage. It is created in memory at execution time.

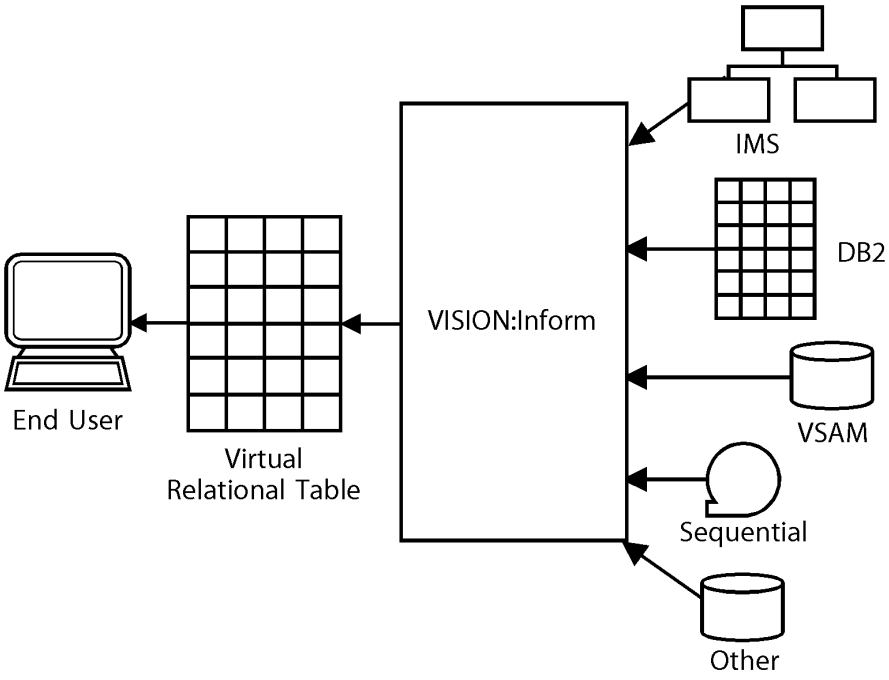

Figure 2-2 The Virtual Relational Table

The virtual relational table is dynamically created by VISION:Inform to simplify data access for end users.

Using the concept of the virtual relational table, all VISION:Inform data extraction requests boil down to simply picking columns from in-memory relational table. The selected columns are then downloaded to the workstation in a simple table format. You do not have to decipher complex hierarchical data structures. VISION:Inform does all that for you behind the scenes.

With VISION:Inform, you can access any combination of DB2, IMS, VSAM, or other databases as if all of the data existed in a single virtual relational table.

Automatic Navigation and Set Operation

In addition to the concept of the virtual relational table, there are other ways in which this unique VISION:Inform feature simplifies the data access and retrieval process for end users.

Providing your end users with the capability to view data as a simple relational table not only simplifies the view of the data, but it also means that VISION:Inform can provide automatic processing of all occurrences and relationships that exist within the physical data.

In the sample hierarchical structure of Figure 2-3, you can use a single data request for two fields, DEPARTMENT and EMPLOYEE_NAME, to automatically retrieve all occurrences of data from the physical data file without even knowing about the hierarchical structure and relationships that exist. All relationships are maintained and all occurrences of data return automatically without you having to know about navigation, relationships, or the physical storage of the data.

In a GUI (Graphical User Interface) environment, the simple act of clicking on DEPARTMENT and EMPLOYEE_NAME returns all the correct information regardless of how the information is physically stored on the host computer. Our example in Figure 2-3, illustrates that two mouse clicks automatically return a set of eight logical records. The data is returned in the proper order with all relationships preserved.

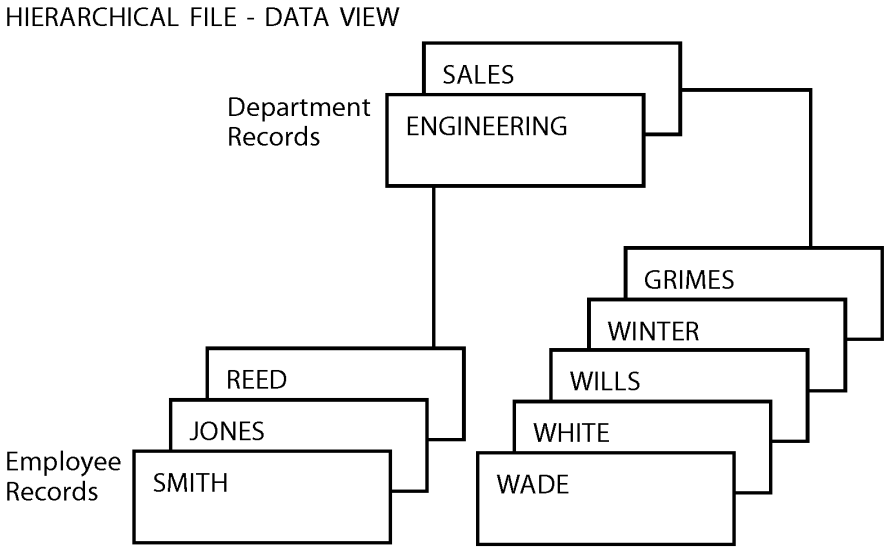

Figure 2-3 Hierarchical View of a Physical Data Structure

As far as the end user is concerned, what is seen is data as if it were from a relational table (Figure 2-4).

Virtual Relational Table - Data View

DEPARTMENT	EMPLOYEE_NAME
SALES	GRIMES
SALES	WINTER
SALES	WILLS
SALES	WHITE
SALES	WADE
ENGINEERING	REED
ENGINEERING	JONES
ENGINEERING	SMITH

Figure 2-4 Relational View of a Hierarchical Data File

With VISION:Inform, you simply state what you want. All data returns to you, in simple table format, with all relationships preserved. You do not need to know how the data is physically organized or where it is physically stored.

The Logical Record

The logical record is another powerful feature unique to VISION:Inform.

Using the VISION:Inform logical record, your database administrator can logically map your existing physical data structures into new logical data structures. This means that you are no longer limited by the constraints of your physical databases.

- You can logically reorganize physical databases, as needed, regardless of the physical type and/or organization.
- You can even customize and control which fields are accessible to a given set of users.

Using the logical record, data organization becomes a dynamic entity, rather than a static entity. You can dynamically manipulate this entity, as needed, to fit your changing business needs.

Using the Logical Record to Simplify Problem Solving

Using logical records means that you can treat the physical organization of your existing data as if it were a dynamic entity rather than a static one. You can use this unique VISION:Inform feature to solve your business problems in a more efficient and effective manner.

Being able to create new logical views of your existing data enables you to always use your data in a manner that is most convenient for solving the problem at hand.

- Logical records provide a method for formulating a simpler conceptual view of a problem.
- They provide independence from the physical structure of the data.
- You can structure and organize the data in the way that you want it stored.

The problem, rather than the physical organization of the data, becomes the driving force in developing an efficient solution. While the results we are looking for are best represented to you in the format of a virtual relational table, the nature of data relationships often makes problem solving a hierarchical process. If the data is organized in the same structure as the problem, solving the problem becomes trivial.

Building a Logical Record to Simplify the Problem Solving Process

DB2 Table — EMPLOYEE

EMPLOYEE_NAME	SALARY	REPORTS_TO

Figure 2-5 DB2 Employee Table

Assume you are the personnel manager for a medium size company. It is morning and the company president has asked that you have a company organization chart by lunch time. You have a single DB2 table called EMPLOYEE. This table contains a row for every employee in the company. Each row contains the following information: Employee Name (EMPLOYEE_NAME), Employee Salary (SALARY), and Name of Employee's Manager (REPORTS_TO).

The organizational chart in question should reflect the personnel hierarchy shown below. Notice that the hierarchy is shown on the left and the requirements for establishing the hierarchy are shown on the right.

President	(Reports to no one)
Vice President	(Reports to the President)
Manager	(Reports to a Vice President)
Technical Staff	(Reports to a Manager)

If you were a programmer, you could solve this problem procedurally by writing a program to process the EMPLOYEE table and produce an organizational chart.

- You would have to make multiple passes of the table, working out and saving all sorts of relationships along the way, to establish the proper hierarchy.
- You would need to deal with the boundary conditions of Vice Presidents or Managers who have no direct reports.

The reason this relatively simple request for data from a simple table structure so difficult to solve is that the problem is inherently hierarchical in nature (as all problems are). But in this case, the data is not hierarchically organized so your solution is not readily available from the data as it is physically stored and organized. To solve this problem, you must write procedural code to establish the correct relationships between the data items.

However, if your data were organized in a hierarchical fashion, as shown in Figure 2-6, the solution is merely to output the data as it exists.

By constructing the proper logical view, you can solve this problem without writing a line of code. You can create a logical data view that will logically reorganize the EMPLOYEE table in such a way that it represents the organizational structure of the company.

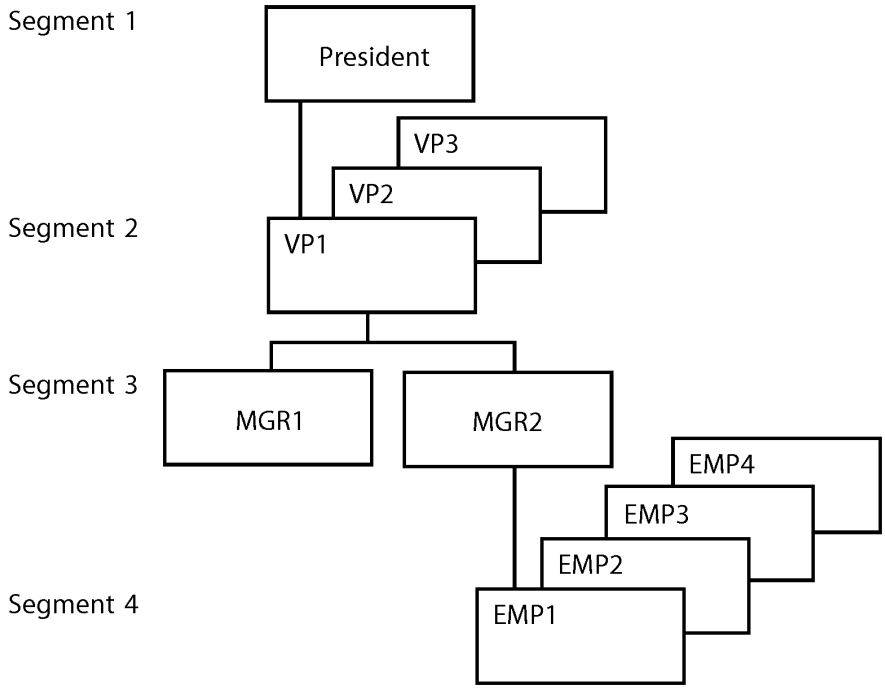

Figure 2-6 Hierarchical Representation of the DB2 Employee Table

To create the logical data view, you will join the EMPLOYEE table with itself four different times, as if it were really four different tables, to simulate a hierarchical structure that mirrors the structure of the desired organizational chart. You will also specify table mapping rules that tell VISION:Inform how to join these tables.

Segment 1: Employee Table, President data
Mapping Rule: Return all rows where REPORTS_TO is blank.

Segment 2: Employee Table, Vice President data
Mapping Rule: Return all rows where REPORTS_TO is equal to the president.

Segment 3: Employee Table, Manager data
Mapping Rule: Return all rows where REPORTS_TO is equal to the vice president.

Segment 4: Employee Table, Technical Staff data
Mapping Rule: Return all rows where REPORTS_TO is equal to the current manger.

Notice that, logically, you now have four structured segments. The structure that you have defined has isolated the data that you want and structured it with the relationships that you need. Thus a single statement of the form:

```
DISPLAY PRESIDENT VP MGR EMP
```

displays the desired organizational chart. No procedural statements are required. The solution to the problem is merely a single statement request for the data that you need.

The File Definition Process

You are able to do this because the data is structured identically to the problem. VISION:Inform lets the data be defined by the means of a simple file definition process. The file definition lets you:

1. Define groups of fields (segments).
2. Establish relationships between the segments.
3. Provide the rules for mapping real physical data into these segments.

Independence From the Physical Database

The logical record is both a simple and powerful capability. It provides you with complete independence from the physical structure of the database. It introduces a new era in problem solving. You no longer solve a problem by trying to fit the solution into the mold created by the physical organization of the data. With VISION:Inform, you solve problems by changing the organization of the data to fit the mold created by the optimum solution.

The Logical Data View

The logical data view is another powerful feature that is unique to VISION:Inform. It can conceptually be thought of as an extension to the DB2 view capability. However, instead of just being able to logically join DB2 tables, you can join separate databases of all types.

- Separate physical databases of any type can be combined to produce a single unified logical view of all the data.
- To the end user requesting data, the logical data view appears as a simple relational table.

To create a logical data view, you simply list the databases that you want to join. Then you specify the synchronization rules for joining these databases. VISION:Inform does the rest making the entire process transparent to you. You view and process the data as if it were really one big relational table.

Automatic Data Transformation Concepts

Logical data views also offer another powerful feature. In addition to giving you the power to join and synchronize databases, the logical data view also provides methods for automatic data transformation.

Suppose that your data has fields containing data represented in feet and inches. However, your company is in the process of switching to the metric system and you must now return metric data to your users. To facilitate this conversion, you can attach a data transformation procedure to a logical data view to automatically convert feet and inches to their metric counterparts whenever the data is referenced. Your end users receive the metric data, never realizing that a transformation occurred. All they know is that they get the data they need, in the format they need, when they need it.

The power of the logical data view is the strength of VISION:Inform. Logical data views enable joining of different databases (such as DB2, IMS, and VSAM) as well as automatically executed data transformations and manipulation procedures.

VISION:Inform Objects and the Definition Processor

VISION:Inform uses four different types of objects to process data retrieval requests:

- File definitions.
- Logical data view definitions.
- Table definitions.
- Procedures.

You can create all of these objects by using an interactive definition generation component called the Definition Processor.

You now know how VISION:Inform uses file and logical data view definitions. The following paragraphs explain how table definitions and procedures fit into the VISION:Inform solution.

Table Definition and Automatic Lookup Concepts

VISION:Inform uses table definitions to provide you with a very powerful and unique automatic look up feature. This feature is best explained by example.

Assume that in your internal company data files, you store a department code rather than storing the entire name of each department within your company. For example, you store "01," instead of "SALES AND MARKETING." Although this saves disk space, it makes reports hard to read as few people can remember which department is represented by which code.

VISION:Inform solves this problem by letting you automatically look up a code value and return a result value that has been associated with that code value.

- You create a table containing the appropriate code values as the table argument entries and the associated replacement values as the table result entries.
- Then you connect this table automatically to file definitions so that when users select the appropriate field, a table lookup operation is automatically performed and the resulting value is returned to you.

The automatic table lookup process is transparent to the end user requesting the data.

Procedural Logic Concepts

Although VISION:Inform provides many powerful and automatic features, there may be special instances where procedural logic is required to solve a particular problem, for instance, automatic data transformation. In these cases, VISION:Inform provides you with a powerful, free-form procedural language that you can use to tie additional automatic processing to your VISION:Inform definitions. Again, this whole process is transparent to the end user.

VISION:Inform is composed of four simple building blocks: table definitions, file definitions, logical data view definitions, and procedures. You can use these objects alone or in combination to provide powerful, seamless data retrieval capabilities to your end users.

The Definition Library

The definition library provides you with an open architecture, central repository for storing all your VISION:Inform objects. Objects in the definition library are stored in source format.

The definition library is a standard OS/390 partitioned data set. You can maintain it by using standard IBM partitioned data set utilities, and you can use your company-standard security packages with it. No special software is required.

The definition process is shown in Figure 2-7.

The Definition Process

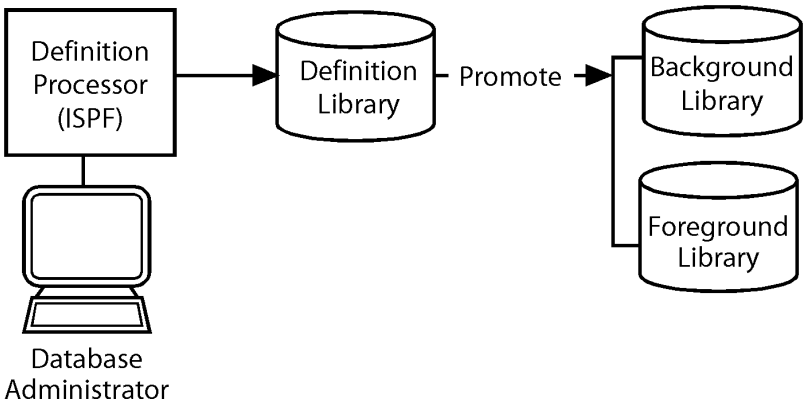

Figure 2-7 The Definition Process

The following text describes how all of these individual components come together to form a cohesive and comprehensive data retrieval environment.

Step 1: Creating Source Objects

The process of data retrieval starts with the database administrator and the creation of VISION:Inform source objects (table definitions, file definitions, logical data view definitions, and procedures). This is generally a one-time process. Once VISION:Inform has been installed and the necessary objects created, you do not have to create or modify them again unless you need changes or enhancements.

The database administrator uses the Definition Processor to create new or modify existing VISION:Inform objects. VISION:Inform source objects are stored in the definition library.

Step 2: Promoting Source Objects

Once a definition is complete and ready to go into production, the definition is promoted from the definition library to the foreground and background processing libraries.

Promoting a source object is similar to compiling a program. The Promote process reads your VISION:Inform definition source statements from the definition library and compiles the source statements into a format that is more efficient for processing. The objects are then stored in the background and foreground processing libraries. Because the background and foreground libraries use and process the definitions in different ways, each library has a format tailored to its specific needs. The different library formats enable VISION:Inform to run in a more efficient manner.

Because your VISION:Inform source objects are not put into production until they are promoted, you can continue to modify and enhance the objects in your definition library without affecting your production environment. When you are ready to put your changes into production, you simply promote the new versions of the objects.

Step 3: Defining User Security Profiles

Before users start accessing host computer data, the system administrator uses the Administration Facilities option from the Main Menu to create user security profiles. These profiles control the files, and even the individual fields within a file, to which each user has access. The creation of user security profiles is generally a one-time process. Once VISION:Inform has been installed and the necessary user security profiles created, you do not have to process the profiles again unless changes are required.

Step 4: Retrieving Data

Once the system administrator creates and promotes VISION:Inform definitions into production, define your user security profiles, and configure the Background Processor JCL, users can, through query processing, retrieve data from the IBM host computer to the workstation.

Step 5: Monitoring System Resources

Once your VISION:Inform system is up and running, the system administrator can monitor the VISION:Inform system activity and make adjustments, as necessary.

Putting it all Together

Several powerful concepts of VISION:Inform have been discussed. Each of these concepts provides powerful capabilities in its own right. Each is designed to solve an aspect of the client/server problem solving process. Taken together, they provide a unique capability. Figure 2-8 illustrates how some of these concepts come together to provide you with the data you want.

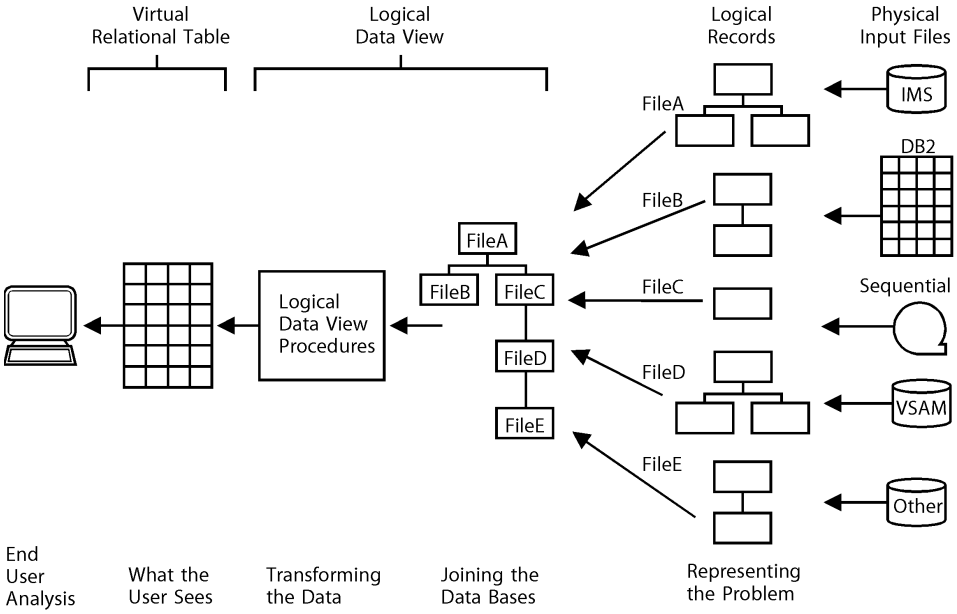

Figure 2-8 The Client/Server Problem Solving Process

Summary

Because of its powerful and unique features, VISION:Inform has many benefits to offer your company. VISION:Inform offers a flexible and easy to use data extraction package.

The following are a few of the many features that make VISION:Inform such a unique and powerful data extraction package.

- Single Centralized Server

  Having a single server provides centralized control of all data extraction requests. Data security and access can be controlled from one location.

- Scheduled Response and Request Batching

With VISION:Inform, you control when data retrieval requests actually get processed. You can also batch similar requests together to minimize the number of database accesses required to fill your user requests.

- The Virtual Relational Table

  You can simplify the end user view of data with VISION:Inform by presenting the data in the form of a relational table. You can retrieve sets of related data automatically if it were stored in a relational table.

- The Logical Record

  With the logical record, you can to logically reorganize physical data into a format that is more appropriate for solving the problem.

- The Logical Data View

  The logical data view provides you with a way to join separate physical files, of different types, to produce a single, unified view of your data. Logical data views also provides you the means to tie procedural processing, such as data transformation routines, to data extraction requests.

- Automatic Table Lookup

  Using VISION:Inform table definitions, you can automatically replace terse codes with more descriptive words or phrases during the data extraction process.

- The Definition Processor

  The Definition Processor is an interactive definition development facility providing your database administrator the means to create all your VISION:Inform objects in a structured, interactive environment.

- The Definition Library

  The definition library provides you with a central, open architecture repository for storing all your VISION:Inform objects (file definitions, logical data view definitions, table definitions, and procedures).

Chapter

3

Understanding VISION:Inform Architecture

VISION:Inform is the host-based server in the Computer Associates client/server architecture. VISION:Inform interacts with the following set of clients:

- VISION:Bridge

  The VISION:Inform subsystem host 3270 client for administration, query preparation, processing, data downloading and routing.

- VISION:Journey for DOS and VISION:Journey for Windows

  The independent PC clients for query preparation, processing and data downloading. See Figure 3-1 for an overview of the basic client/server environment.

Client/Server Environment

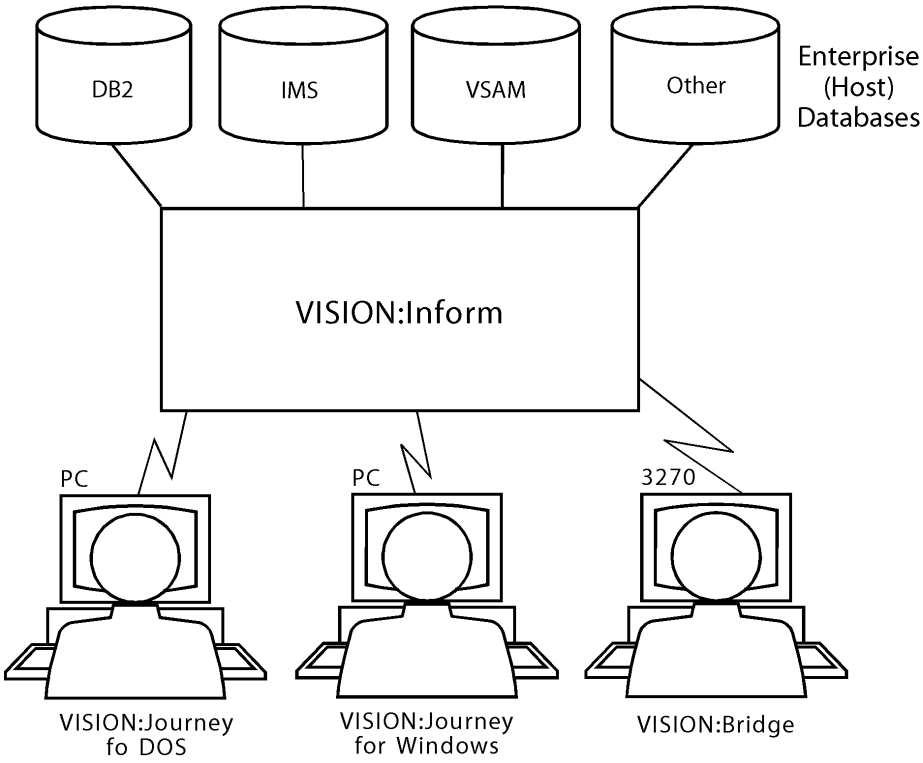

Figure 3-1 The Client/Server Environment

VISION:Inform Components

VISION:Inform and the client software products work together to provide workstation users with a cooperative processing facility to access virtually any file or data management system supported on IBM hosts.

With the client software product, workstation users can request VISION:Inform to extract selected subsets of data, optionally summarize them, and download them to the workstation to use with other applications.

VISION:Inform is made up of a number of integrated components. Your installation administrators set up these components.

The primary components of VISION:Inform are:

- Foreground Processor
- Background Processor
- Definition Processor
- Communication File
- Foreground Library
- Background Library
- Definition Library

Typically, the system administrator installs the product, creates the online VISION:Inform profiles, defines security access to other profiles and data files, and creates queries as needed.

Typically, the database administrator sets up the Definition Processor, runs certain utilities, and populates the definition library, background library and the foreground library.

Typically, the host-based 3270 end users and client-based end users create and submit queries, and download the data output from those query requests.

The relationships among these components are shown in Figure 3-2.

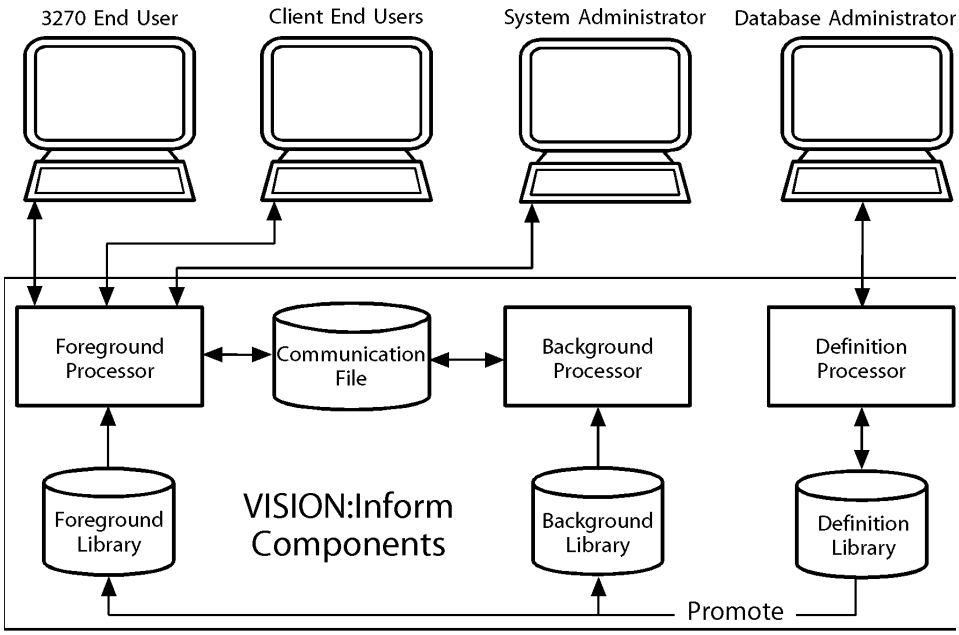

Figure 3-2 VISION:Inform Components

VISION:Inform operates in the host computer environment and communicates with client products running on workstations and other client platforms.

Foreground Processor

The Foreground Processor provides two major functions.

- It provides the system administrator with facilities to define security constraints and monitor resource usage.
- It also acts as a communications interface to the workstation.

To the online environment in which it operates, the Foreground Processor is just another application program. It conforms to the same design constraints and operating considerations of other application programs running in that environment.

Users make requests, called queries or tasks, to the Foreground Processor through the client product. The system administrator sets up security procedures the using the Foreground Processor. Throughout these functions, the Foreground Processor interactively checks for internal consistency. As a security function, it checks the profile to ensure that the user can access the data requested or perform the action requested.

The Foreground Processor writes the submitted query or task to the communication file where it remains until a Background Processor, for its processing class and data view, becomes active.

Background Processor

The Background Processor (known as the Batch Processor in releases prior to 3.1) operates in the batch processing areas of your operating system. It contains a program which receives the submitted client requests for data, batches them, if possible, with other requests, and then processes them either individually or in batches.

The Background Processor retrieves the data and manipulates it according to the request specifications.

The Background Processors processes queries and tasks in the communication file by class within database sequence.

Upon completion of processing, the Background Processor returns output to the communication file. The data remains on the communication file until the client requests delivery through the Foreground Processor.

You can group queries (from a 3270 interface) and a tasks (from a workstation) into 14 different groups or classes. Each class can be processed by a separate Background Processor. Thus, you can have up to 14 different Background Processors which can execute at different times with different priorities.

You can set up Background Processors to process different sets of databases, different groups of users, or to satisfy different degrees of priority of response (such as every minute, every hour, or overnight). Databases named in a Background Processor must also be specified (or inherited) in the online-referenced VISION:Inform profile.

The computer operator can control the Background Processors by releasing the execution JCL at predetermined times.

Definition Processor

The Definition Processor provides you with a way to develop and maintain your VISION:Inform definitions. VISION:Inform definitions include:

- Table Definitions
- File Definitions
- Logical Data View Definitions
- Procedure Definitions.

The Definition Processor operates in the ISPF program development environment. In this interactive environment, you can develop your VISION:Inform definitions quickly and easily.

The Definition Processor validates and saves all VISION:Inform definitions in the definition library. The definition library is a partitioned data set which holds all of your VISION:Inform definitions.

The online execution environment does not reference the definition library. Thus, you can add and modify items without impacting any other users of VISION:Inform. Definitions do not become active until they are transferred, or promoted, to the background and foreground libraries.

The Promote process compiles selected definitions from the definition library and copies them to the background and foreground libraries for use by VISION:Inform.

Foreground Library

The foreground library is an execution file containing definitions used by the Foreground and Background Processors. It contains all of the user profiles, definitions, queries, and statements (incomplete, unvalidated queries).

- User profiles provide the system security.
- Definitions permit the Background Processor to recognize and access databases and files. VISION:Inform promotes definitions from the definition library to the foreground library using the Promote Process Utility.
- Queries specify which fields are extracted from databases and logical databases.

Background Library

The background library is an execution file containing all of the definitions used by the Background Processor. VISION:Inform promotes definitions from the definition library to the background library using the Promote Process Utility.

Definition Library

The definition library is a standard, open-architecture, partitioned data set used by the Definition Processor. It contains all of your VISION:Inform source definitions. When the definitions are ready to be put into production, the system administrator promotes them to the background and foreground libraries using the Promote Process Utility.

Communication File

The Foreground Processor and the Background Processor use the communication file to transmit information between them. Queries and tasks submitted by the client software are stored in the communication file and, if possible, batched when processed. The communication file provides:

- A means for queries and tasks to be transferred from the workstation to the Background Processor.
- A storage medium for queries and tasks to await processing.
- A repository for host extracted data to reside for subsequent downloading, viewing, routing, printing, and deleting.

Chapter

4

Understanding Files

This chapter discusses files and file definition concepts.

- A file is a named set of records that you store and process.
- A file definition is a description of file structure, characteristics, and contents.

You create and validate file definitions by using the Definition Processor and save them in the definition library.

File Concepts

When you are using VISION:Inform, you make requests for data from a logical record.

It is important to understand the distinction between a logical record and a physical record.

Physical Record

A physical record is a representation of data as it resides on a physical disk storage device or some other external medium (such as tape). This is physical data, because it is where you keep the data.

Logical Record

A logical record is a representation of the record that is used by the system from which to extract data. A logical record is not necessarily the same as a physical record.

Some database access methods require that the data that is taken from the disk and placed in memory is an exact copy. In this case, the logical record equals the physical record. However, other database managers (notably relational database managers) do not demand that the logical data be exactly the same as the physical data.

VISION:Inform Logical Records

VISION:Inform expands upon this concept and provides you the capability to define logical records which are dramatically different from the physical records. With the VISION:Inform process, you can define logical records which accumulate information from a single database.

Logical Data Views

Logical records from different databases can then be combined together into a logical data view (see Chapter 5, *Understanding Logical Data Views*). This provides you with a simultaneous logical view of data from a variety of databases.

If more than one database is involved, the definition process has two steps. First, you define the logical record associated with each database, and then you define the means of linking the databases.

- A logical record is a logical representation of data extracted from a single physical database.
- A logical data view is a join of logical records that have been taken from different databases. For more informatoin about logical data views see Chapter 5, *Understanding Logical Data Views*.

The Logical Record Process

The logical record process involves defining the logical, or in-memory, record as well as providing the rules for mapping the physical data into the logical record.

- When the logical record is the same as the physical record, no explicit mapping instructions are required.
- When the logical record is dramatically different from the physical record (as with relational database managers), specific mapping instructions are provided in order to transfer the data from the physical to the logical.

In the subsequent discussions, when file definitions are mentioned, we are referring to logical files and logical file definitions.

In defining the logical record, you provide three types of information:

- Descriptions of each of the logical fields which can be used by the client systems.
- Relationships between sets of fields (segments).
- Rules for mapping the physical data into the logical form.

Physical Data Versus Logical Data

With some database managers, the logical data must be the same as the physical data; with others, the relationships between segments is implicit by position within the record.

With VISION:Inform, you can use most of the file or data management systems supported on IBM host systems.

- You can use data stored in various formats (such as fixed length, variable length, or undefined length), and retrieved through various access methods and database management systems (such as DB2, IMS, and VSAM).
- You can also form logical data view structures containing information from different databases, relational tables, and files (described in Chapter 5, *Understanding Logical Data Views*).

Despite the variety of file types supported, you use the same definition procedure to describe all files available for processing. (See the *Advantage VISION:Inform Definition Processor Reference Guide* for detailed specifications for defining files.)

File Definition Terms

In a logical file definition, you describe the data structure of the file. The following terms are used when discussing file definitions:

- Fields are the smallest logical unit of data.
- Segments are logically related groups of fields.
- Records are groups of logically related segments.

Flat Record

The simplest kind of record that you can define is a flat record. A flat record contains one segment.

The CUSTOMER file shown in Figure 4-1 is an example of a file that contains flat records. Three records from the CUSTOMER file are shown in Figure 4-1. Each record contains the same five fields in the same location. The fields are named CUSTOMER_NUMBER, NAME, ADDRESS1, ADDRESS2, and TELEPHONE.

CUSTOMER_NUMBER	NAME	ADDRESS1	ADDRESS2	TELEPHONE	
0573	APEX	211 MYPLACE	ANYTOWN	555-7321	Record 1
0791	BAKER	762 YOURPLACE	SOMETOWN	666-8742	Record 2
0934	CHARLIE	666 SOMEPLACE	ATOWN	777-9313	Record 3

Figure 4-1 An Example of a Flat CUSTOMER File

Fields in this logical record are fixed in length. This means that information in the same field in each record (that is, ADDRESS1 in Figure 4-1) is limited to the maximum field length.

Hierarchical Record Structures

You can define a record with more than one segment. This type of record is called a hierarchical record, because it establishes a parent-dependent relationship between the segments, where one segment is a parent to one or more segments. Figure 4-2 shows a hierarchical CUSTOMER file record.

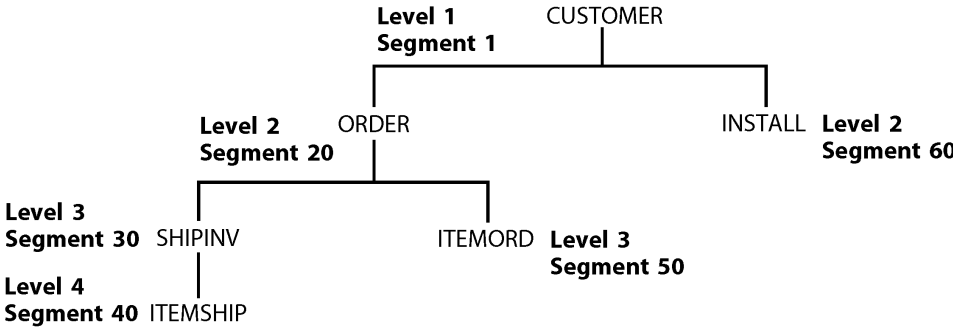

Figure 4-2 An Example of a Hierarchical CUSTOMER File

Hierarchical record structures are an extremely powerful problem solving technique particularly when converting them to logical relational tables.

One of the best ways to solve a problem is with a hierarchical structure. The organizational chart illustration in Figure 2-6 is an example of this powerful capability. When the data structure matches the problem structure, problem solving is trivial.

With VISION:Inform, you can have the best of both worlds. You can store your data in relational tables with all their advantages of simplicity of maintenance and control. You can use this same data as a hierarchical record for problem solving.

Segments

A hierarchical file can contain records with up to 99 different kinds of segments, and each segment can contain different information in different fields.

In the hierarchical CUSTOMER file shown in Figure 4-2, the CUSTOMER segment is the parent of the ORDER and INSTALL segments, and the ORDER segment is parent to the SHIPINV and ITEMORD segments. The SHIPINV segment is the parent of the ITEMSHIP segment.

- The CUSTOMER segment is also referred to as the root segment. A root segment is the highest level segment in a hierarchical record, and there is only one occurrence of a root segment in each record.
- The ORDER segment and its dependent segments contain information about orders placed by the customer.
- The INSTALL segment contains information about the customer installation.

Fields in Segments

Segments are further divided into one or more fields. This is similar to the way in which the records of the CUSTOMER file shown in Figure 4-1 are divided into fields.

- For example purposes, the fields shown in Figure 4-1 can comprise the level 1 (root) segment of the hierarchical CUSTOMER file in Figure 4-2.
- The ORDER segment in Figure 4-2 contains fields for the order number, date, sales person, due date, invoice generation, and order completion.
- The INSTALL segment in Figure 4-2 contains fields for the installation number, contact person, operating system, and product release number.

Using Level Numbers and Segment Numbers

You use level numbers and segment numbers to describe the hierarchical relationship between segments in the logical record. In Figure 4-2, the level and segment numbers are annotated next to each segment.

The levels in the hierarchy specifying the parent-dependent relationship between the different segment types are numbered from one to nine.

- The root segment, which contains the essential data that occurs only once in a record, is always level 1.
- Level numbers 2 through 9 are assigned to dependent segments according to their relationship to their parent segment.

The level number of a dependent segment is always one greater than its parent segment.

Assigning Level Numbers

When you define a hierarchical file, you assign level numbers that adhere to the following conventions:

- Number the levels consecutively from top to bottom.
- There must be one and only one segment at level 1 (the root segment).
- The level number of each segment must be one greater than the segment immediately above it in the hierarchy.

Assigning Segment Types Unique Numbers

You must assign each segment type in your file a unique segment number. Segment types, their relationship to one another, and position in the hierarchical record structure are defined using segment numbers. Each segment type within a record must have a number from 1 to 99. When you assign segment numbers:

- Number segments from top to bottom down each leg of the hierarchy; work through the legs from left to right.
- Number each segment type according to its relative position in the record structure. Each segment number must be larger numerically than the segment immediately above it, but smaller than that of the segments to the right, even if lower in the hierarchy.
- Segment numbers need not be consecutive. Incrementing them by some multiple means that you can add segments later, if you need them. It is a good practice to leave a gap of 5 or 10 between segment numbers.

Fixed Occurring Versus Variably Occurring Segments

In a hierarchical file, the segments you define can occur once or multiple times. Each record in a file can contain many occurrences of a single segment type. For example, in a single customer record there can be many orders for items.

- When a segment type occurs the same number of times in every record, the segment is a fixed occurring segment.
- A segment in which the number of occurrences varies depending on the record is a variably occurring segment.

Figure 4-3 represents a record from the CUSTOMER file with variably occurring segments. For example, in Figure 4-3, the number of items in the ITEMORD segment varies based upon the order for items described by the ORDER segment.

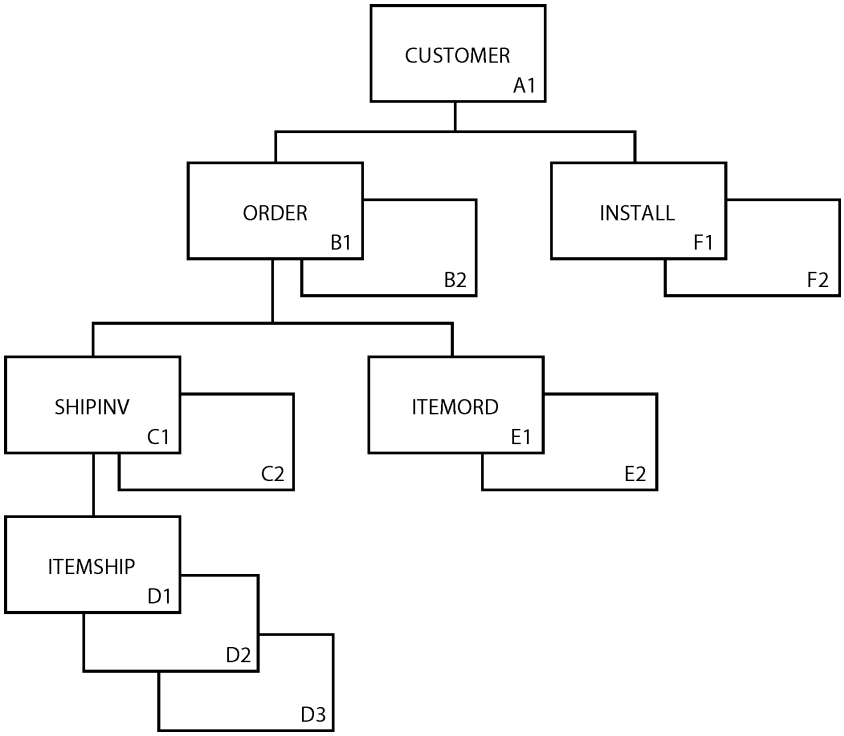

Figure 4-3 The CUSTOMER File Record with Variably Occurring Segments

Requirements of Logical Records

Logical records have certain requirements that are independent of the requirements of any particular database manager. Observe the following rules:

- Every segment must have at least one field defined as a key field. Depending upon the database manager, this field may or may not correspond to a field which is actually a record key of the physical file.

  In some database managers, such as DB2, there are no keys inherent in the structure of the database. Nevertheless, you must identify at least one field in the logical record as the key field for the segment.

- The length of a segment is determined by the last byte defined in the definition of the logical segment.

  If the database manager has constraints that demand that the logical record segment be the same as the physical segment, then you must define the last

byte of the segment correctly. You can achieve this by defining a dummy field of a single byte to establish the correct size of the segment.

- If a hierarchical logical record is defined, each of the levels of the hierarchy must increase by one. Further, number the segments in order down the left most branch of the record structure, moving across to the right most branch.
- In defining a hierarchical structure, the dependent segment can be defined as having either a fixed number of occurrences or a variable number of occurrences.

  If the segment is defined as having a variable number of occurrences, the parent segment should contain a count field which provides the number of occurrences for each segment type.

  – In some database managers, this count field is essential for identifying the structure of the physical record.

  – In other database structures, such as IMS and DB2, the count field is not essential for defining the structure but is maintained by VISION:Inform.

  A good practice is to define a count field in the parent segment for each dependent segment that will have a variable number of occurrences.

IMS Databases

You can define an IMS database the same way you define any other file, but there are some special considerations described in this section.

IMS Record Structures

An IMS database record is arranged in a hierarchical structure See Hierarchical Record Structures for more information. Each record in an IMS database contains one root segment and up to 255 segment types.

- The root is a parent segment, and there can be one or more dependent segments below the root segment. The root segment is always referred to as level 1 in a file definition.
- The immediate dependents of the root segment are the level 2 segments. Segments at all levels can be parents to other segments at lower levels.

An IMS database can contain up to 255 segment types and 15 levels of dependency; however, the VISION:Inform file definition supports at most 99 segment types and 9 levels. (See IMS Segment Considerations for more information.)

Keyed or Unkeyed IMS Segments

A segment in an IMS database can occur any number of times. In IMS databases, space is never reserved for future segment occurrences. An IMS segment can be keyed (uniquely or nonuniquely) or unkeyed.

The VISION:Inform file definition requires a key. Keyed segments are accessed in ascending EBCDIC order.

Multiple occurrences of the same key value are retrieved in the order of its insertion in the record. When you access unkeyed segments, the segment is retrieved according to the order of its original insertion into the record. The RULE parameter in the DBD (database description) determines where in a chain of occurrences a segment is to be inserted for unkeyed segments.

Variable Length Segments

IMS also permits databases to contain variable length segments. When you describe an IMS variable length segment in a file definition, the length of the segment must reflect the maximum size of the segment, including the 2 byte binary length field in position 1 of the segment.

IMS Control Blocks

Defining an IMS database depends upon information from several IMS control blocks. The first of these, the DBD (Data Base Description), defines your database to IMS. It contains the following information:

- The organization, access method, DBD name, file name/DD name, and record size or block size of the database.
- The names and lengths of each of the segments in the database.
- The position of each segment within the hierarchy.
- The name, length, location, and type of the segment key fields, and any other fields that can be used to access segments.

The record size and block size in the DBD do not indicate the total size of the IMS record. The record size and block size in the DBD indicate the amount of data transferred in one physical I/O operation.

The PSB (Program Specification Block) describes your program's view of the database. The PSB consists of one or more PCBs (Program Communication Blocks).

PCBs (Program Communication Blocks)

Each PCB describes the program's view of the database. IMS does not require that you name all the segments in the PCB; you only name those database segments that your application can access. The PCB also describes how your application can

access a database. You can use the PCB to limit the view of a segment to only certain fields. When you limit the PCB view to certain fields, your application program only sees the defined fields in the I/O area, not the entire segment.

The segments you describe in the PCB, however, must maintain the hierarchical structure of the database as described in the DBD. This means you cannot skip any database level, and you must maintain the left to right sequence of segments.

For more detailed information about the DBD, PSB, and PCB control blocks, see the *IBM IMS/ESA Utilities Reference* manual.

Figure 4-4 shows a sample of the DBD and PCB used to describe the CUSTOMER database.

```
     DBD         NAME=CUSTOMDB,ACCESS=(HISAM,ISAM)
     DATASET     DD1=CUSTOMER,DEVICE=3350,OVFLW=CUSTOMOV
     PRINT       NOGEN
*
     SEGM        NAME=CUSTOMER,PARENT=0,BYTES=45
     FIELD       NAME=(CUSTNI,SEQ,U),BYTES=5,START=1,TYPE=C
     FIELD       NAME=CUSTNAME,BYTES=30,START=6,TYPE=C
*
     SEGM        NAME=ORDER,PARENT=CUSTOMER,BYTES=39
     FIELD       NAME=(ORDERNO,SEQ,U)BYTES=5,START=1,TYPE=C
*
     SEGM        NAME=SHIPINV,PARENT=ORDER,BYTES=98
     FIELD       NAME=(SHIPNO,SEQ,U),BYTES=4,START=1,TYPE=C
*
     SEGM        NAME=ITEMSHIP,PARENT=SHIPINV,BYTES=24
     FIELD       NAME=(ITEMSHIP,SEQ,U),BYTES=7,START=1,TYPE=C
*
     SEGM        NAME=ITEMORD,PARENT=ORDER,BYTES=37
     FIELD       NAME=(ITEMORD,SEQ,U),BYTES=7,START=1,TYPE=C
*
     SEGM        NAME=INSTALL,PARENT=CUSTOMER,BYTES=81
     FIELD       NAME=(INSTNO,SEQ,U),BYTES=4,START=1,TYPE=C
*
     DBDGEN
     FINISH
     END

PCB    TYPE=DB,POS=M,KEYLEN=0028,PROCOPT=AP,DBDNAME=CUSTOMDB
SENSEG  NAME=CUSTOMER
SENSEG  NAME=ORDER,PARENT=CUSTOMER
SENSEG  NAME=ITEMORD,  PARENT=ORDER
SENSEG  NAME=INSTALL,PARENT=CUSTOMER
PSBGEN  LANGUAGE=ASSEM,MAXQ=1,PSBNAME=CUSTPSB
```

Figure 4-4 The DBD and PCB for the IMS CUSTOMER Database

Notice that the DBD contains six segments and the PCB contains only four of those segments (with the hierarchical order maintained), thus limiting the view to something less than the entire database.

IMS Control Blocks and File Definitions

Creating a file definition for an IMS database depends on the information in the IMS control blocks. The DBD and PCB contain the necessary information to specify your file definition.

The DBD name specified in both your DBD and PCB must be specified in the DBDNAME entry on the DL/I File Definition panel.

All of the IMS access methods are supported by VISION:Inform. FASTPATH databases are supported under either of the following conditions:

- If PARMBLK OPTMODE parameter is set to 2 or 3 so qualified Segment Search Arguments (SSAs) are not generated.
- If dynamic optimization is turned off.

See the *Advantage VISION:Inform Installation Guide* for your environment.

Logical record structures are also supported. No matter what access method is used to access the database, the file definition makes the access method completely transparent to the application; it is just a selection on the File Type panel.

Figure 4-5 shows the relationship of the DBD, PSB, and PCB control blocks to the file definition.

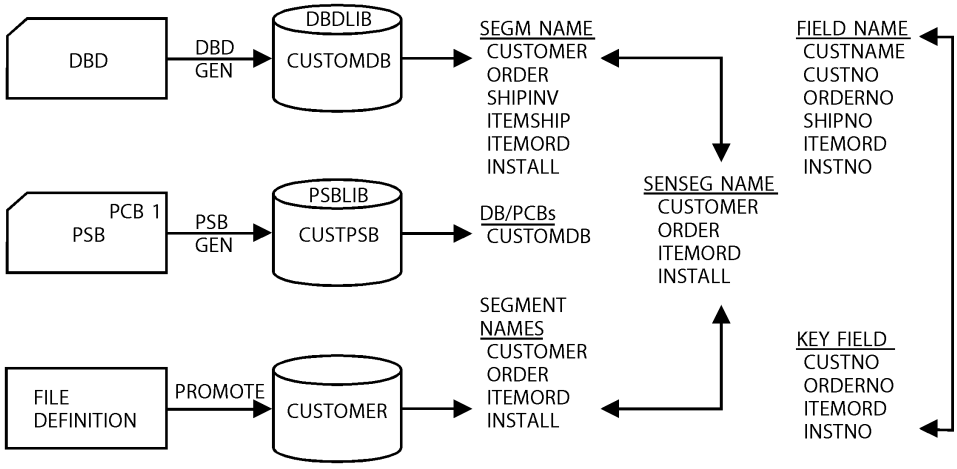

Figure 4-5 The Relationship of the DBD, PSB, PCB, and File Definition

The DBD shown in Figure 4-4 describes all of the segments in the database, but the PCB describes only those segments in the DBD that the application can access. Therefore, the file definition only describes those segments defined in the PCB.

VISION:Inform requires you to define only that portion of the database described in the PCB. The only other requirement is that you limit the PCB view of the data to nine levels and 99 segment types.

Relationship of the DBD to the Hierarchical Structure

The DBD determines the hierarchical structure of the database you define.

For example, in Figure 4-4 the CUSTOMER segment is defined before the ORDER segment in the DBD.

- The CUSTOMER segment must specify a lower segment number than the ORDER segment in the file definition.
- Similarly, if the ORDER segment is the parent of the ITEMORD segment, the level number of the ORDER segment in the file definition must be lower than the ITEMORD segment.
- The PARENT entry in the DBD determines the parentage of segments in the file definition and how level numbers are assigned.

See Hierarchical Record Structures for more detailed information about level and segment numbers.

IMS Segment Considerations

This section contains information that you need when you define segments in an IMS database. You must adhere to the following segment considerations:

- Each segment defined in a PCB SENSEG statement must correspond to a file definition segment. Every segment named on a PCB SENSEG statement must be defined on a Segment Definition panel in your file definition.
- Segment numbers are assigned sequentially, and each one must be a higher value than the preceding one.
- The hierarchical structure of the database must be maintained.
- The segment size in the file definition must match the segment size in the DBD.

Also be aware that segments in IMS databases can occur a variable number of times.

IMS Segment Length

The length of an IMS database segment is computed when you promote your definition.

- The length is computed based upon the starting location and length of the defined fields for that segment.
- The computed length must agree with the length defined to IMS for the segment.
- If the length of the segment in the file definition does not match the IMS segment length, unpredictable results can occur during processing.

When you use a physical DBD (HISAM, HSAM, HIDAM, or HDAM), the BYTES keyword of the SEGM statement explicitly states the segment length. If the last byte of a segment has not been defined as part of a segment field, you must, as a minimum, define the last byte as an arbitrary 1-byte field.

When the source for a logical segment is a single segment, the logical segment length is defined in the BYTES keyword of the physical DBD defining the segment. If a logical DBD is used, segment length is defined in the SOURCE keyword of the SEGM statement. A SOURCE keyword naming two segments (a logical dependent and either its physical or logical parent) joins these segments to form a single defined segment. The amount of each segment used in this concatenated segment depends on the specifications in the SOURCE statement.

Generally, concatenated segments contain the key of the logical/physical parent followed by the logical dependent segment. If you specify the DATA operand in the SOURCE statement describing the parent, the entire parent segment is going to follow the logical dependent.

If field sensitivity is used, the file definition segment can consist of the sensitive fields concatenated, rather than the entire segment.

IMS Segment Keys and Search Fields

IMS does not require segments to be uniquely keyed. It also does not require search fields to be uniquely named across segments in the DBD. VISION:Inform, however, requires that each segment defined in the file definition have a unique segment key and name.

- If the key field name is not unique in the DBD, you can assign an alias name.
- You can also use non-keyed fields as search fields if they are defined in the DBD.

You indicate the field is a search field on the Field Definition panel. See the *Advantage VISION:Inform Definition Processor Reference Guide* for detailed specifications for field definitions.

Naming Segment Key Fields

The names defined in the file definition as segment key fields and search fields must be the same names that appear in the DBD FIELD statements. Their correct spelling is important, because they are used in segment search arguments (SSAs) in your applications.

You can have multiple segment search fields if they are defined in the DBD with a FIELD statement.

Assume Signed Packed Fields

Note that VISION:Inform always assumes that packed fields are signed. Because of this assumption, unexpected results can occur when IMS keys or search fields contain unsigned packed fields. IMS does a byte-by-byte comparison of the field in the Segment Search Argument (SSA) to the field in the database. If the compare finds an F sign in the data, they are not equal, because there is a C sign in the SSA. Qualified SSAs are built by VISION:Inform.

IMS Segment Ordering

You can also specify segment ordering in file definitions for IMS databases; however, you must exercise caution when using segment ordering for numeric keys.

- IMS uses EBCDIC sequences for maintaining keyed segments regardless of the field type.
- VISION:Inform uses algebraic sequence.

Unexpected results can occur if your file definition specifies ordering on an IMS segment that is defined with a numeric key when the sequence of the stored segment is not algebraic. For example, if a segment is defined in the DBD with a 1-byte numeric key (SEQ=U) then the database contains the following hexadecimal occurrences:

00, 0F, 70, 7F, 80, 8F, F0, and FF.

These occurrences are stored in the database by IMS in EBCDIC collating sequence. If this same IMS segment is defined to be ascending in the file definition, the values are not interpreted in algebraic order by VISION:Inform. Instead the values are interpreted as follows:

+0, +15, +112, +127, -128, -113, -16, and -1.

IMS Processing Considerations

VISION:Inform does not require segments to be ordered for standard retrieval. For this type of processing, you can specify segments in any order that accurately reflects their occurrence in the database.

For memory optimized retrieval, ordering is required unless only one hierarchical leg is defined in the file definition.

Standard Processing

To process a database with standard processing (not memory optimized), you can create a file definition that describes all the segments in the DBD instead of just those in the PCB. (The DBD structure must not have more than nine levels and 99 segment types.)

With standard processing, if you reference a field in a segment that is not defined in the PCB, the reference is treated the same as a reference to a non-existent segment. The size of the IMS record determines the amount of memory required to process with standard processing. Standard processing reads the entire record into memory. It retrieves all of the segments defined in the PCB; therefore, the larger the IMS record, the more memory required to process it.

Memory Optimized Processing

When you use memory optimized processing, you must only define those segments that are defined in the PCB. With memory optimized processing, if you reference a field in a segment not defined in the PCB, it results in a unexpected status code from IMS.

Matching Field Type and Length

You define the fields within each segment on Field Definition panels. The type and length attribute of each defined field must match the type and length attribute defined in the field statement in the DBD.

For fields not defined on the FIELD statement in the DBD, the type and length must also match the characteristics of the actual data.

IMS Secondary Indexing

Secondary indexing is a feature of IMS. A secondary index provides an entry point to a database through a field other than the primary root key. The segment containing the entry point field is the target segment. The segment containing the index key field is the source segment.

The source segment must be either the same segment as the target segment or a dependent of it. IMS requires that the target segment must never be a dependent of the source segment.

Processing an IMS Secondary Index

If you use secondary indexing, you need a different file definition than the one you use for normal (primary index) processing. There are two reasons for this.

- First, the new file definition describes the secondary data structure.
- Second, the XDFLD NAME field from the DBD through which secondary indexed access occurs must be defined as either a real or virtual key field in the secondary root segment.

You can process all secondary index structures, but you must define the structure and its key.

- If your target segment is the primary root, the primary structure is identical to the secondary index structure. The only exception is the root segment key. A new file definition that specifies the new root key (the XDFLD SRCH field) should be used.
- If the target segment is not the primary root, you must have a file definition that maps the secondary index structure. The PCB for the secondary index structure describes the segment hierarchy and sensitivity.
- If the target segment is a dependent segment, the PCB must also reflect the segment hierarchy when it is inverted.

Inverted Structure

In the inverted structure, the target segment becomes the root. The segments on which the target is dependent in the primary structure becomes the leftmost hierarchical leg, in inverted order. Figure 4-6 shows an example of an inverted secondary index structure.

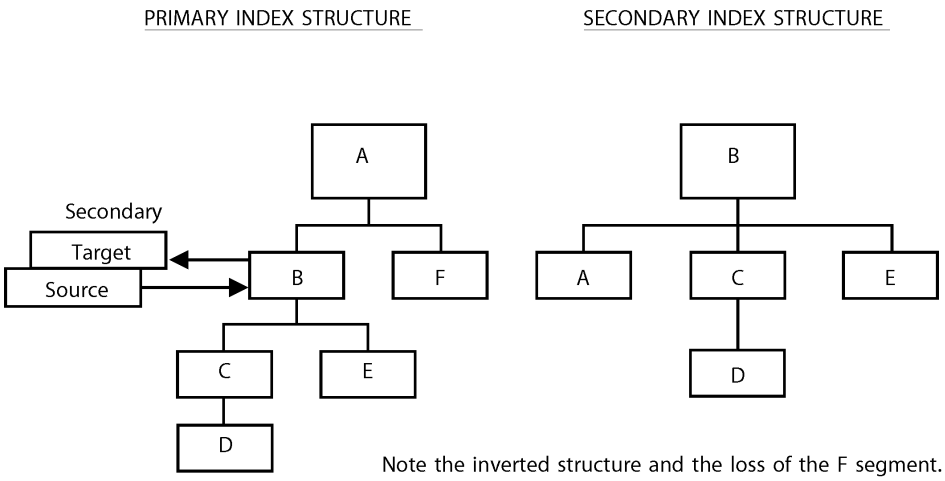

Figure 4-6 An Inverted Secondary Index Structure

Secondary Index Structure

In the secondary index structure, the XDFLD SRCH field in the DBD becomes the root key for the secondary structure.

- If the XDFLD SRCH field is in your target segment and it consists of contiguous fields, define that field as the new key in your file definition. Specify a new key with a 1 in the SEGMENT KEY entry on the Field Definition panel.
- If the source and target are not the same segment, or if the XDFLD SRCH field consists of non-contiguous fields, then you must define a virtual key field in the new root segment in your file definition. Specify a virtual key with a V in the SEGMENT KEY entry on the Field Definition panel. You define the virtual key field as the root segment key. See Virtual Key Fields for information about the use of virtual key fields in file definitions.

With the exception of the segments on which the target segment is dependent, the remaining secondary segment structure is the same as that of the primary structure.

A segment retains the same name, length, key fields, search fields, and hierarchical relationships.

VISION:Inform is able to perform retrieval processing, both standard and memory optimized, on databases with secondary indexes.

Virtual Key Fields

You define a virtual key field using the parameters from the XDFLD statement for the secondary structure. Virtual keys are treated just as any other database segment key. The virtual key field does not physically exist on the database, it is developed by IMS during retrieval.

The virtual key field is made available as though it is part of your root segment for all processing.

You define a virtual key in order to provide a key field for the following conditions:

- A segment in a secondary index structure whose target (root) segment does not contain a key field.
- The secondary key is made up of non-contiguous fields.

Secondary Indexed Structures

This section discusses different examples of secondary indexes. These examples describe the following:

- Various combinations of target segment, source segment, and XDFLD SRCH field.
- Definition requirements.
- Processing restrictions.
- IMS restrictions.

The example in Figure 4-7 shows a primary and secondary index structure that has the same record structure. In this example, the target field is part of the primary root segment. The DBD for this structure can contain single, contiguous or non-contiguous XDFLD SRCH fields. To process this record structure, create a new file definition for the secondary structure or override the key field in the primary structure.

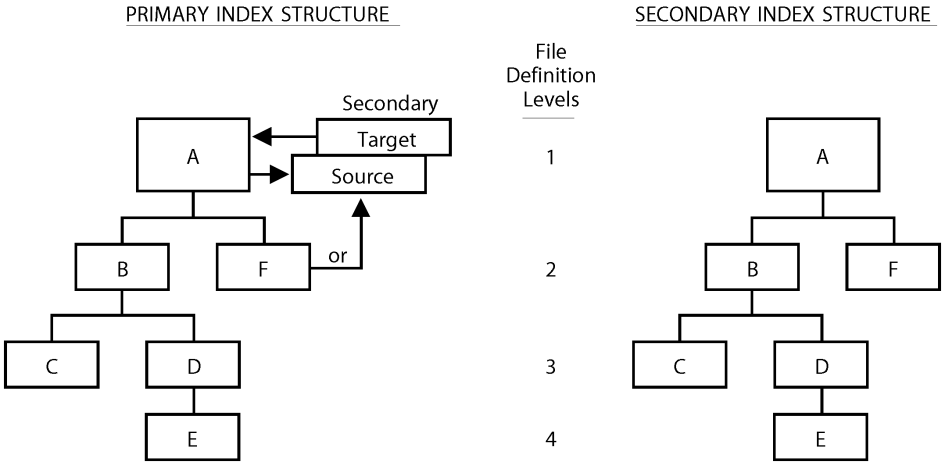

Figure 4-7 A Secondary Index with Target and Source Fields in the Root and a Dependent Segment

Although the secondary structure in Figure 4-7 is identical to the primary structure, a new file definition should be created that defines the new root key. This new file definition must specify a virtual key field from the source (XDFLD) fields as the record key.

If you intend to process this structure, you can use the same file definition, but if the key is not changed, you cannot use the file in a logical data view.

Figure 4-8 shows a primary and secondary index structure with the target and source field in the B segment of the primary structure. The DBD for this structure can contain single, contiguous or non-contiguous XDFLD SRCH fields.

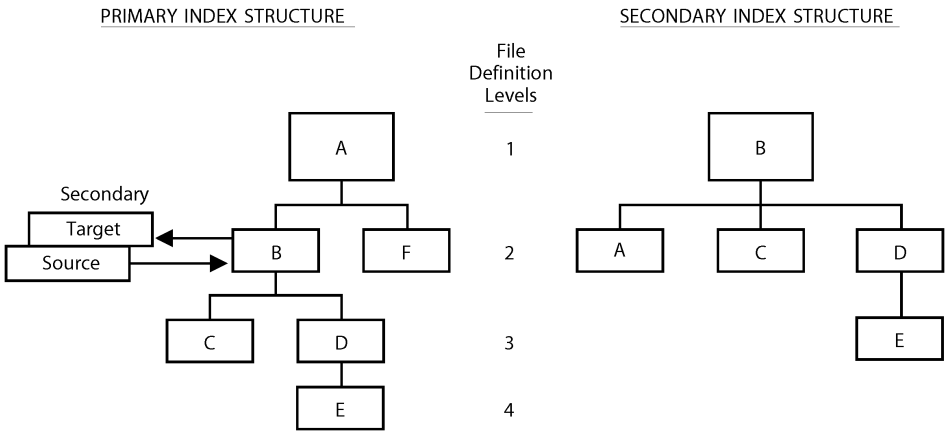

Figure 4-8 Secondary Index Structure with the Target and Source Field in a Dependent Segment

You can use the secondary index structure in Figure 4-8 for the full range of processing, if you do the following:

- Create a new file definition for the secondary index structure.
- Make sure the segment level numbers in the new file definition match the inverted structure.
- Define a virtual key field from the source (XDFLD) fields as the record key.
- Promote the new file definition with a new name.

Once the file definition for the secondary structure in Figure 4-8 is promoted, you can perform full processing.

- The example in the top portion of Figure 4-9 shows the same primary and secondary index structure as in Figure 4-8.
- The bottom portion of Figure 4-9 shows the same primary structure as in the top portion of Figure 4-9.

The difference between the structures in the upper and lower portions of Figure 4-9 is that the lower portion structure contains a target field in the D segment and a source field in the E segment. These target and source fields create the secondary structure shown in the lower half of Figure 4-9.

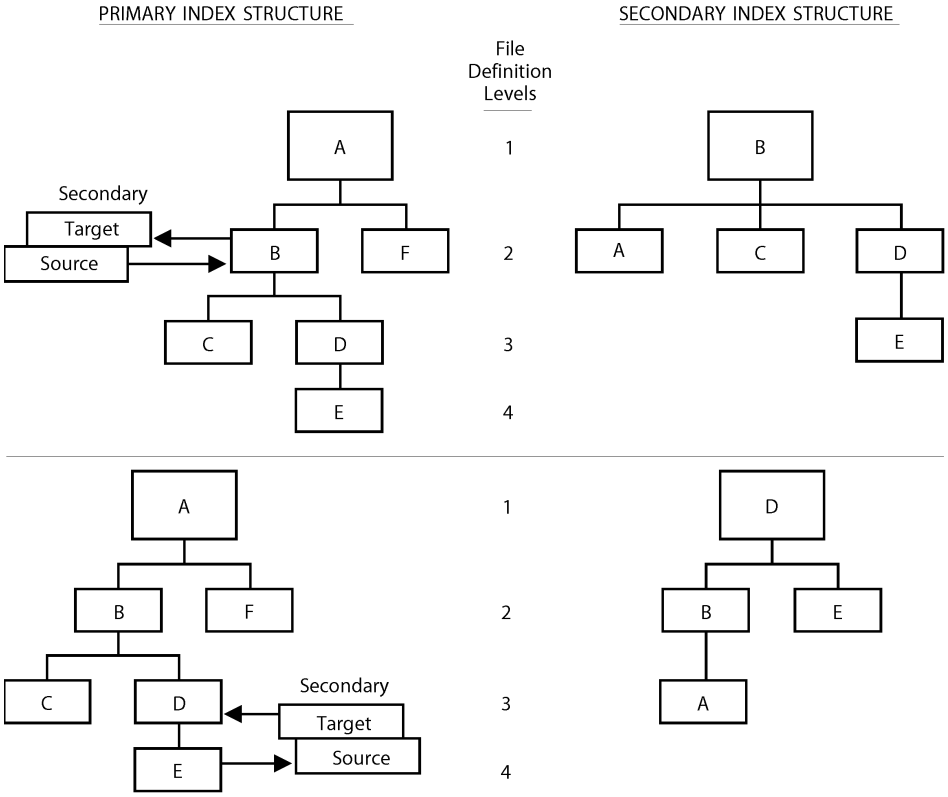

Figure 4-9 Secondary Index Structure with the Target and Source Fields in a Lower Level Parent and Its Dependent

You can use the secondary index structure in the lower portion of Figure 4-9 for the full range of processing, if you do the following:

- Create a new file definition for the secondary index structure.
- Specify a virtual key field for the record key field using the XDFLD SRCH field for the record key.
- Make sure the segment level numbers in the new file definition match the inverted structure.
- Save and promote the new file definition with a new name.

Preparing a Secondary Structure Definition

The steps below are generalized guidelines you can use for preparing a file definition for a secondary structure. As you define your own secondary structures, you might have to modify these steps. For detailed specifications for defining file definitions, see the *Advantage VISION:Inform Definition Processor Reference Guide*.

1. Examine the PCB listing for the database you want to define. If the PCB statement includes a PROCSEQ parameter, the database has a secondary index. Create a file definition for a secondary index.
2. The PROCSEQ parameter indicates a secondary data structure. The PCB (not the DBD) reflects the hierarchical structure of the database. Your file definition must map this structure exactly.
3. Note the segment names in the PCB SENSEG statements. You will need them in the file definition.
4. Define non-key fields.
5. Determine the key field names, locations, lengths, and field types in the usual manner for all segment types, except the root of the secondary data structure.

When PROCSEQ is specified, the root segment of the secondary data structure is always accessed through a secondary index. The record key field you describe in the file definition must be the XDFLD of the index. You can determine the characteristics of this XDFLD in the following manner:

1. Examine the listing of the DBD named in the DBDNAME or NAME parameter of the PCB statement.
2. Locate the SEGM statement that names the root segment of the secondary data structure. This SEGM statement describes the index target segment type for the structure. It is followed by at least one LCHILD statement before any other SEGM statements appear in the listing.
3. From these LCHILD statements, locate the one that names the secondary index for the structure. The secondary index name is the second subparameter of the NAME parameter of the LCHILD statement. For example:

   LCHILD NAME=(PARTPTR, PARTINDX),...

   PARTINDX is the secondary index name and is the name specified by the PCB PROCSEQ parameter
4. Locate the first XDFLD statement following this LCHILD statement. The NAME parameter of the XDFLD statement specifies the XDFLD name to use as the record key in your file definition.

5. Determine whether to define a real key or virtual keys by examining the SEGMENT and SRCH parameters of the XDFLD. (The SRCH parameter names the fields that make up the XDFLD.)
6. Define a virtual key if the name specified by the SEGMENT parameter differs from the segment name in the preceding SEGM statement.

   In this case, the index source segment is a dependent of the index target segment and the XDFLD is in the index source segment. Also, define a virtual key if the key is non-contiguous.
7. Define a real key if the SEGMENT parameter is omitted or if it specifies the same segment name as in the preceding SEGM statement.

   In this case, the index source and index target segments are identical. The XDFLD is in the index source segment and the XDFLD is made up of contiguous fields.

A virtual key does not specify a start location. To determine a real key's start location, do the following:

1. Locate the leftmost (or only) field name specified by the SRCH parameter. Then find that field's START value among the FIELD statements for the segment. This is shown in Figure 4-10.

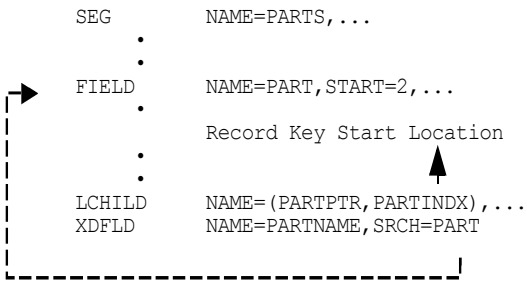

Figure 4-10 An Example for Determining the Record Key Start Location

2. Determine the length of the record key by combining the length of all fields specified in the SRCH parameter of the XDFLD statement.

   The virtual key field length is the combined length of the individual fields. To determine the individual field lengths look at the BYTES value for each field in the index source segment type (which is the segment named by the SEGMENT parameter of the XDFLD statement). This is shown in Figure 4-11.

3. You must always specify the field type for the record key as a character string. This is true regardless of the characteristics of the individual fields which make up the XDFLD.

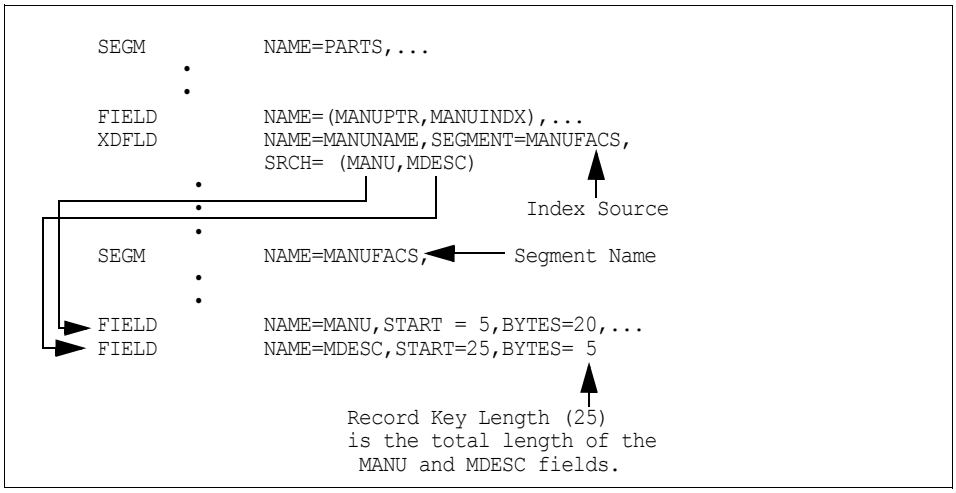

Figure 4-11 An Example for Determining Field Lengths

When your PCB specifies INDICES for a root segment, you can prepare and create your file definition exactly as described in the preceding guidelines. There is only one difference between processing INDICES at the root level and PROCSEQ. Using INDICES causes sequential retrieval of root segments in the primary process sequence instead of in the secondary process sequence. Direct access uses XDFLD.

When your PCB specifies INDICES for lower level segments, some processing problems may arise. Although you can define an XDFLD as a lower level segment key, it is not recommended. You should define normal segment keys for all lower level segments for which INDICES is specified.

Optimizing IMS Database Access

You can optimize accesses of segments of data from an IMS database by the use of SSAs.

- VISION:Inform automatically generates SSAs for the access of the data based upon the content of the queries submitted to it.
- Because the SSAs are dynamically generated and vary from execution to execution (depending upon the number and type of different requests in the run), this form of optimization is called dynamic optimization.

Writing Optimization Procedures

You can also override the dynamic optimization and provide your own optimization including within the logical data view a procedure of type P. For information on procedures, see Chapter 7, *Understanding Procedures*. The procedure is converted into an SSA.

The rules for writing optimization procedures are:

- Specify only one procedure for any given database segment.
- Specify only a single IF statement with the logical expression for the procedure.
- The single IF statement may only reference database fields from a single IMS segment.
- The single IF statement may not contain more than eight relational operators (for example, EQ, NE, LT, LE, GT, GE).
- The left-hand operand of each relation must be a field which is defined to VISION:Inform as a key field or a search field. In turn, each key field or search field must be defined within the corresponding IMS database definition (DBD).
- The right-hand operand of each relation must be either a constant or a previously defined temporary field.
- The left-hand operand of any relation may not include any partial field specifications.
- The logical expression cannot contain any parentheses.
- Do not provide a procedure for the root segment of a secondary file.
- The IMS Program Control Block (PCB) must be sensitive to all segments in the hierarchy when storage optimization is specified for the given file.

Examples:

PROC1

IF 0.STATE EQ 'CA' AND 0.SALARY LT 35000

This statement generates an SSA for the appropriate IMS segment in the root or primary file of the logical data view (logical file DBFILE0).

PROC2

IF 2.DEPTNO EQ 'CO2', 'D37', 'E15' AND 2.SEX EQ 'F' AND 2.SAL LT T.MALEMIN

This statement generates an SSA for the appropriate IMS segment in secondary synchronized file DBFILE2 of the logical data view.

For information on logical data views, see Chapter 5, *Understanding Logical Data Views*. For information on procedures, see Chapter 7, *Understanding Procedures*.

Static Optimization

When you supply a procedure to provide the optimization rules for IMS, you are in effect providing your own SSA. Since this SSA is being provided as part of the logical data view, it is a predefined SSA. This form of optimization is called static optimization.

- It is only possible to use static optimization when dynamic optimization is not active. VISION:Inform provides the capability to turn off dynamic optimization. For information on making dynamic optimization inactive, see the *Advantage VISION:Inform Installation Guide* for your environment.
- Any attempt to use static optimization in the same run with dynamic optimization will result in the run being terminated.
- Under normal circumstances, you can let VISION:Inform use its dynamic optimization to improve the performance of the database extraction.

If you want to use static optimization, you must provide SSAs by means of type P procedures in the logical data view and disable dynamic optimization.

Relational Tables

You define relational tables to VISION:Inform as logical records. Logical records reside in-memory and are derived from one or more (physical) relational databases.

- With files such as IMS databases and VSAM files, you must define a one-to-one link between the physical file and the logical file.
- However, with relational logical records, you define records according to your application needs. The records are temporarily constructed in memory as a result of a mapping process that takes place during application execution.

In general, you define logical records that best suit the solution of the problem under investigation.

The end user will not be aware that the data originated from relational databases and will not be aware of the logical structure that you defined. The end user simply requests data by field name, and that data is retrieved and made available.

You can define a single relational table as a logical record consisting of just a root segment (such as a flat file), or you can define multiple relational tables as a logical hierarchical record.

The definition that you create and promote determines the way in which the relational tables are mapped into logical records.

Figure 4-12 shows how several relational tables can be logically related (indicated by broken lines) through fields containing common values.

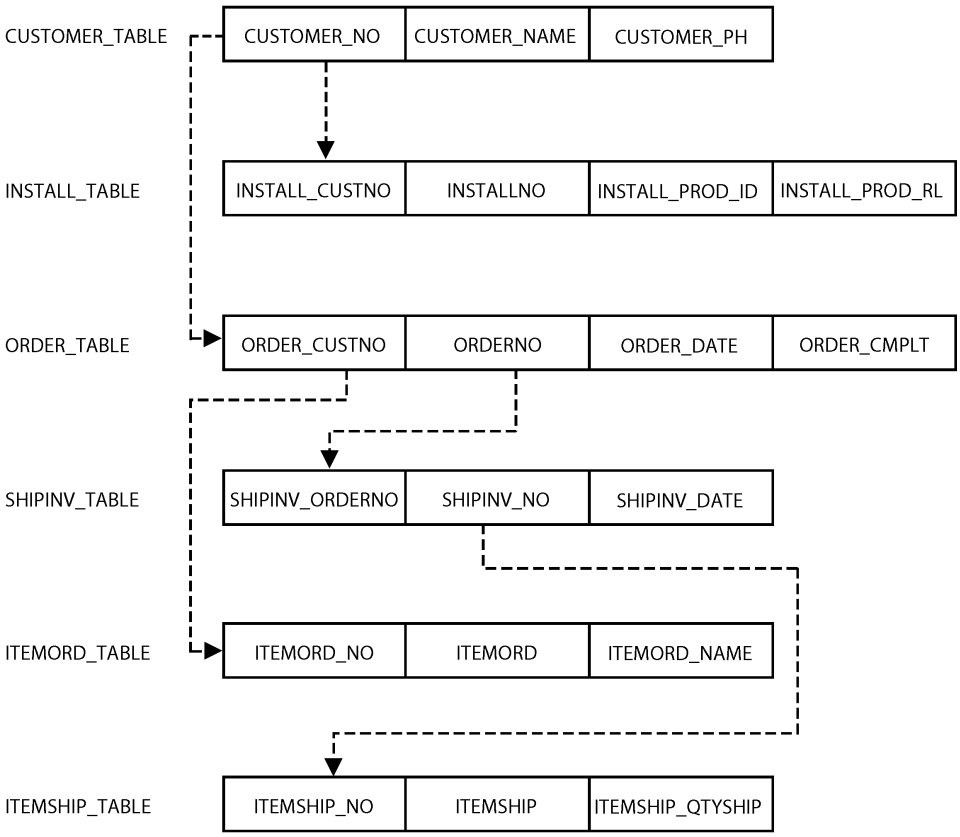

Figure 4-12 Relational Tables with Logically Related Fields

Figure 4-13 also shows the relational tables after being mapped into a logical hierarchical record. In the logical record, each segment represents a table. Segments and logical record structures are described in more detail in the following section.

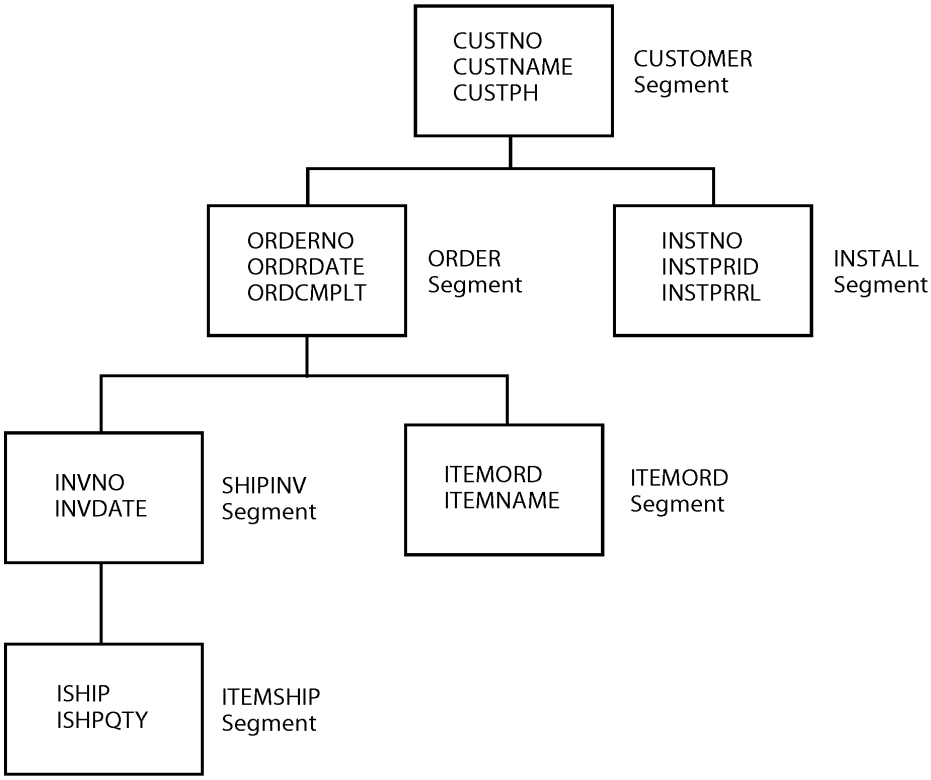

Figure 4-13 A Sample Logical Record Structure

You can define relational logical records to be used as root files and/or as synchronized files.

This section describes the special considerations for defining logical records. For detailed specifications for defining logical records, see the *Advantage VISION:Inform Definition Processor Reference Guide*.

Mapping Relational Tables to Logical Records

The data within a relational table is organized in rows (horizontally) and columns (vertically). This type of organization is similar to any two-dimensional table. You can think of a single relational table as a flat file, and each row in it as a record in the file. You can define and process a relational table the same as any other single level flat file. When you define a relational table in this way, DB2 is used as the access method for the file. Each selected row in the relational table becomes a record in the file. When you retrieve columns from a relational table, you can retrieve them in any order you want.

Defining Relational Tables in a Hierarchy

When you define multiple relational tables into a logical record, you define them in a hierarchy. A hierarchy is a collection of paths of related data. VISION:Inform offers you the advantage of defining hierarchies that best suit the problem you are trying to solve with your application.

Unlike IMS databases, which are hierarchically structured, relational tables are not hierarchical in nature. But, by using the logical record definition you can easily define a group of relational tables in a hierarchy.

To construct a hierarchical structure, define the relational tables with one or more related columns. You decide how you want to structure the hierarchy when you define the logical record. The hierarchy you define is determined by the way you define relationships between segments. You have complete control over the creation of the hierarchical structure, and its eventual processing. A logical hierarchical record can contain up to 99 relational tables or views, which are mapped into segments in the logical record.

Establishing Relationships Between Segments

You establish the hierarchical structure of the logical record by including, in your definition, conditional statements that establish relationships between the segments. A segment in a logical record is a mapped relational table. The segments in the structured hierarchy are organized in a parent-dependent relationship.
See the section File Concepts for additional information about parent-dependent relationships.

In the logical record in Figure 4-13, the following relationships exist:

- CUSTOMER parent of ORDER and INSTALL.
- ORDER parent of SHIPINV and ITEMORD.
- SHIPINV parent of ITEMSHIP.
- INSTALL, ITEMORD, ITEMSHIP no dependents.

In the logical hierarchical record, the rows within each relational table are included in the logical record as repeated occurrences of the segment. You can retrieve a row of a relational table based upon a variety of conditions.

Conditions for Row Selection

The establishment of these conditions provides both the parent-dependent relationship between segments, as well as filtering the data to be processed. You define the conditions for row selection on conditional statements that you include in your logical record definition.

The conditional statements determine which rows from which tables are to be retrieved. At application execution time, all the relevant rows from all the relevant relational tables are retrieved. They are retrieved by parent segment, and within each parent segment, all of its dependent segments are retrieved. Segments are retrieved by satisfying the relationship criteria on the conditional statements.

Providing Conditional Statements that Define Fields

You provide the conditional statements in the logical record definition along with the definition statements that define the fields that compose each segment. The columns in the relational tables that you want to use in your application must be defined as fields in the logical record. However, relational columns that are not defined as fields in the logical record can still be used in conditional statements.

In the logical record, you define the relational tables as segments and the table columns as fields. In the CUSTOMER table in Figure 4-12, there are three fields (such as CUSTOMER_NO, CUSTOMER_NAME, and CUSTOMER_PH). When the CUSTOMER table is mapped into the logical record shown in Figure 4-13, the CUSTOMER segment contains the following fields:

- CUSTNO
- CUSTNAME
- CUSTPH

In a relational table, the order of the fields is unimportant, because the fields are referenced by their names, not by their position in the table. DB2 requires column names to be unique within the table. When you define a logical record, you define a position location for each field within a segment, but you reference each field by its name. The segment names and field names that you define in the logical record must be unique.

Logical Record Segment Considerations

You define a logical hierarchical record with segments just like other file types.

Identifying the Relational Table Name

For segments in a logical hierarchical record, you must also identify the relational table name that is mapped into the segment. The relational table name can be from one to 18 characters, but the segment name can only be one to eight characters. The table name of the relational table is used to map the table into the logical record. The table name may be fully qualified as AuthorizationId.Tablename.

Specifying the Segment Definition

In your segment definition, you must:

- Name segments.
- Associate relational table names to segments.
- Establish the relationship of segments to one another in the hierarchy.
- Specify the segment sequence as ascending or descending.
- Number the segments the same as any definition type.

Using Logical Expressions

The occurrences of a segment within a logical record correspond to each occurrence of a row within the relational table.

- To identify which rows of the relational table should be treated as segment occurrences in the logical record, your file definition must include a logical expression specified on the Definition Processor Logical Relationships panel.
- Every segment that you define in a relational logical record, except the root segment, must have one or more logical expression statements that define the relationship.
- In these logical expressions, you specify the conditions under which dependent segments belong to a parent segments. You can include constants or names of fields to be used in conditional tests. A comparison is made between the fields or constants and the columns in the relational table.

You can use the following conditional operators:

- EQ (equal)
- GT (greater than)
- GE (greater than or equal)
- LT (less than)
- LE (less than or equal)
- NE (not equal)
- NL (null)
- NN (not null)

You can use these operators to test for multiple conditions by combining conditional statements with AND and OR connectors. An unlimited number of conditional statements can be combined in a single test.

Retrieving Rows in Relational Tables

Rows in relational tables are not organized in any kind of sequence as are data segments in some of the other file types. The sequence of the rows in a relational table is unimportant when you define it as a logical record and process it sequentially.

When you process the logical record sequentially, the rows of the relational table are retrieved by the fields defined as segment keys in the order specified on the Field Definition panel (such as ascending, descending, or unordered). Keys do not exist in relational tables; however, logical record definitions require each defined segment to have a unique name and at least one field identified as a segment key.

Define logical record fields that are ordered as key fields. To indicate that the field is a key field in your logical record definition, specify a key value from 1 to 9 (1 is the highest and 9 is the lowest record key) when you define the field.

Logical Record Fields

When you define the fields in the logical record, you assign a unique name that you must use in all future references to the field. You define fields in the logical record almost identically to the way you define fields in other files.

- Each field that you want to use in each segment is defined on the Field Definition panel. The fields are mapped into the logical record from columns in the relational table.
- On the Field Definition panel, you must provide the mapping rule for transferring data from the table into the logical record.

Mapping Relational Table Columns to Logical Record Fields

The columns in relational tables are similar to fields in any other file. Each column in a relational table contains data which can be retrieved and processed by some application. The data in the column is one of the entities that make up a row. When a relational table column is mapped into a field in a logical record, the data from the column becomes a logical field which is one of the entities that make up a segment.

In most other file types, the smallest addressable unit is usually a record or segment; however, columns in the relational table rows are stored as separate entities that can be individually referenced. This added flexibility makes it unnecessary to retrieve or define all the columns in a row as segment fields in the logical record definition. In your logical record definition, you only define and retrieve those fields (columns) that are actually required for your application.

The defined fields in the logical record are filled with data from the columns in the relational tables. For this reason, you provide the mapping for the relational table in your definition.

Mapping Fields

A segment is a logical item, so it does not need to contain all of the fields (columns) of the relational table that it represents. This differs from VSAM and IMS where the logical and the physical segment lengths must be the same. This is illustrated in Figure 4-14.

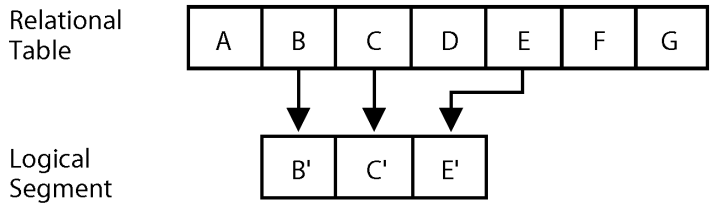

Figure 4-14 Subsetting the Table Columns

Figure 4-14 shows the mapping (the rules for movement) of physical fields B, C, and E to logical fields B', C', and E'. You specify this mapping by the entry on the Field Definition panel which identifies the name of the field (column) that is the source of the data. When defining the logical field, you merely identify which physical field is to supply the data to fill it.

Automatic Mapping

This automatic mapping capability is more powerful than merely mapping a single physical data field into a logical data field.

- You can map scalar functions such as:

  1.05 * SALARY
  BUDGET + VARIANTS

- You can use any legitimate scalar function supported by SQL for the purpose of mapping into the logical record.
- In addition to scalar functions, you can use the DB2 built-in column functions. For example:

  MAX (SALARY)
  MIN (SALARY)
  AVG (SALARY)
  SUM (BUDGET)

  You can use any legitimate function.

Thus, a logical database can be comprised of fields that are derived from the physical data. A logical data field can actually be a copy of the field, the result of an arithmetic expression, or a compound field obtained by summarizing over several rows of the table.

Creating Fields

You can define fields in the logical record that do not exist in relational tables. For fields that do not exist in relational tables, you do not specify a relational column name on the Field Definition panel. The field is defined in the logical record, but data is not transferred into it from the relational table.

Overlapping Fields

Overlapped fields can exist in the logical record definition, but they cannot be stored as overlapped fields in the relational table.

Relational tables do not support all of the field types that you can define in the logical record (such as zoned decimal). There are also some slight differences in the way that various field types are supported.

Automatic Field Conversion

For your convenience, an automatic conversion is performed when a relational table column is mapped into a logical record. A field conversion, when it occurs, can slightly increase the amount of time your application takes to process.

Dynamic and Static Optimization

VISION:Inform automatically constructs WHERE clauses for DB2 depending upon the query specifications of the end users.

Batching and Constructing WHERE Clauses

The VISION:Inform Background Processor has the capability of batching together many different requests from different users to access data from the database. Thus, VISION:Inform passes the database only once for the purpose of extracting data for several users. VISION:Inform analyzes the different requests for data from all users batched together and based upon these requests constructs suitable WHERE clauses which result in optimal performance.

Dynamic Optimization

Since the different requests batched together from different users vary from run to run, this analysis is performed each time prior to passing the database. This form of optimization is called dynamic optimization.

Providing Your Own WHERE Clauses

However, there are times when you know the characteristics of your database better than VISION:Inform can deduce from the nature of the queries provided. Thus, VISION:Inform provides you with the capability to provide your own WHERE clause with DB2. You provide WHERE clauses to VISION:Inform in the logical data views.

- When defining a logical record generated from a DB2 database, you can display a panel on which you can provide the necessary WHERE statements.
- You can display this panel when you define the synchronization rules between the DB2 logical record and any other logical record.

Static Optimization

When you provide WHERE clauses, you are defining the optimization to be used within the logical data view. This form of optimization is known as static optimization.

Disabling Dynamic Optimization

Static optimization and dynamic optimization are mutually exclusive. Therefore, in order to be able to use static optimization, you must disable dynamic optimization. The process of disabling dynamic optimization is described in the *Advantage VISION:Inform Installation Guide* for your environment.

Providing Alternate Names

When you define fields in logical records, you can provide a primary name, which is up to eight characters in length, and an alternate name, which can be up to thirty characters in length. The alternate name capability is provided so that end users, who are accustomed to DB2 column names or to longer COBOL names, can continue to use them when requesting data from the client systems.

Using the Primary Name in the Logical Data View

When using field names anywhere within the logical data view, you must use the primary name in the following circumstances:

- When you are specifying the fields which will synchronize the logical records, you must use the primary names.
- If you reference any fields from a logical record within your WHERE statement, you must use the primary names.

VSAM Files

You can access VSAM files just like any other file. The definition process for VSAM files is the same as for any other file you define, except for some special considerations as described in this section. For information on for defining VSAM files, see the *Advantage VISION:Inform Definition Processor Reference Guide*.

You can define VSAM files to be used by VISION:Inform root files and synchronized files.

VSAM Record Structures

A VSAM record can be arranged in the same type of structures described in the section File Concepts. VISION:Inform supports two types of VSAM file organizations:

- KSDS (Key Sequenced Data Set).
- ESDS (Entry Sequenced Data Set).

Alternate Indexing (AIX)

VSAM also supports alternate indexes. An alternate index provides an alternate entry to a KSDS or ESDS file.

- For KSDS files, the key field in your root segment must be in ascending order by key value and each key must be unique; only one key field is permitted.
- For ESDS files, an arbitrary root key field must be defined even though ESDS files do not normally require a key.

By using an alternate key, you can access ESDS files sequentially.

You can also directly access ESDS files using alternate index path processing. Change the file definition to use an alternate key and save it as a KSDS file.

VSAM Cluster Definitions

To create a file definition for a VSAM file, you need information from the VSAM cluster definition. Figure 4-15 shows a sample VSAM cluster definition for the VSAM CUSTOMER file. The cluster definition defines the characteristics of your VSAM file to the VSAM access method. Many of these same characteristics must also be defined in the file definition.

- The RECORDSIZE parameter in the cluster definition contains the average and maximum lengths of the data record.
- The maximum record length must be specified in the BUFFER SIZE entry on the file definition panel.
- The maximum record length also influences the control interval specification in the cluster definition.

For more detailed information about cluster definitions, see the *IBM Access Method Services* manual.

The cluster definition indicates the type of VSAM file. The INDEXED or NONINDEXED parameters indicate that the file is KSDS or ESDS. For INDEXED, the KEY parameter in the cluster definition describes the length and relative location of the record key.

In the sample cluster definition shown in Figure 4-15, the record key is located in position one of the record (relative location zero) and its length is five bytes. The file definition for a KSDS must match the key specifications in the cluster definition.

```
DEFINE CLUSTER -
     (NAME (PSO.CBT.CUSTOM) -
     FILE(CBTCUST) -
     RECORDS(25 5) -
     VOLUMES(DOSV01) -
     RECORDSIZE(1000 2000) -
     INDEXED KEYS(5,0)) -
     DATA -
      (NAME(PSO.CBT.CUSTOM.DATA) -
      CONTROLINTERVALSIZE(2048)) -
     INDEX -
      (NAME(PSO.CBT.CUSTOM.INDEX)) -
CATALOG(AMASTCAT)
```

Figure 4-15 A Sample VSAM Cluster Definition for the CUSTOMER File

VSAM Alternate Indexing

A VSAM alternate index is a VSAM feature that provides an entry to a key sequenced data set (KSDS) or entry sequenced (ESDS) data set through a field other than the primary root key. Through an alternate index path, you can process sequentially or directly.

You can use an alternate index path by cataloging a file definition with the alternate key field as the record key. Another alternative is to process the alternate index as a file.

If you want to process the alternate index as a file, you must also create a file definition for the alternate index. In the alternate index definition, specify the following:

- Option AIX on the File Type panel.
- The control interval size in the BUFFER SIZE entry on the File Definition panel.

Chapter

5

Understanding Logical Data Views

Logical data views (LDVs) join different databases or files into one logical file. This capability is an extension of the DB2 View concept. The DB2 View enables you to join relational tables; VISION:Inform logical data view enables you to join all forms of databases.

In addition, logical data views support procedures (PROCs). When you execute these procedures against the data, they manipulate or transform the data prior to being made available to the users.

Logical Data View Concepts

Logical data views establish relationships among two or more databases or files.

- You can include up to 10 logical files in a logical data view.
- A logical data view can consist of any combination of existing IMS databases, IMS logical databases, DB2 tables, and other files.
- You designate one file as the primary file and designate the other files as synchronizing files.

Synchronizing Key Field

The files are joined by synchronizing them based on the contents of a synchronizing key field. This field can originate in the primary file, in one of the synchronized files, or a field computed within a PROC (temporary field). Once the logical data view is defined, you can use data from fields in any of the files as though the fields are in a single file.

Synchronization

The logical data view is completely transparent once you define it; however, a certain amount of automatic processing takes place to construct the logical data view. This processing is called synchronization.

Synchronization joins records in a primary file with records in another file or files by matching fields to keys in the synchronized file. When a match occurs, the matching record is read into memory.

Thus, a logical data view consists of:

- A set of related files from which data is simultaneously extracted.
- Procedures that provide optimization rules for the access of IMS databases or procedures that provide transformations or computations on the extracted data.

Using Logical Data Views

You can use logical data views in the following ways:

- To process multiple data files of various types that are otherwise unrelated.
- To process synchronized files that are not ordered in the same sequence as the primary file.
- To read a subset of records in a synchronized file based on selected primary file records.
- To use the synchronized file as a large table and perform a table lookup based on the contents of the synchronizing field.
- To process multiple primary file records against one synchronized file record.
- To use different queries to access different records from the same synchronized file.
- To provide data transformations (or screening) prior to making it available to the user running the query.
- To optimize the performance of IMS.
- To make dynamic modifications to the synchronization process.

Logical Data View Structure

When you join logical records from different databases, the result produces a composite record of data from each of the databases involved. This is the logical data view record.

In a logical data view, you designate one databases as the primary file. This is essentially the controlling file.

Figure 5-1 shows the structures of two files used in a sample logical data view.

- The CUSTOMER file is the primary file. The CUSTOMER file has six segments.
- The ITEM file is the secondary or synchronized file. The ITEM file has two segments.

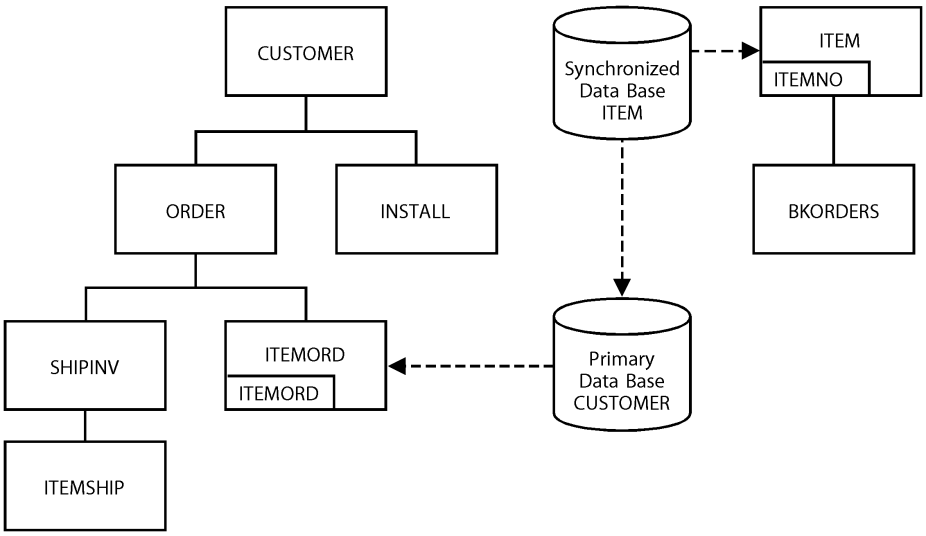

Figure 5-1 The Structures of the CUSTOMER and ITEM Files

The synchronizing field in these files contains a unique item number for each of the company's products. This item number resides in the root segment field ITEMNO in the ITEM file and in the ITEMORD field in the CUSTOMER file.

When a match occurs between the two fields, the two physical files are joined. As a result, a single record is composed in memory.

Figure 5-2 shows what the structure of the logical data view for these two files.

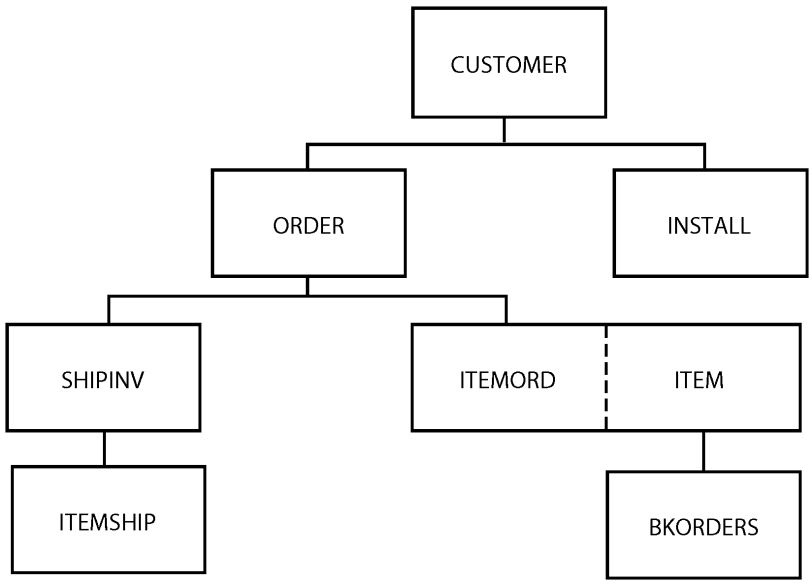

Figure 5-2 The CUSTOMER/ITEM Logical Data View Structure

Defining a Logical Data View

To define a logical data view:

- List the files to be part of the logical data view.
- Identify the synchronization fields (the field that the record key of the secondary files must match in order to be part of the logical data view record).
- List the PROCs to be used to help create the logical data view.

The formatted source listing for this logical data view is shown in Figure 5-3.

```
                              ******************************
09/30/01                      * LOGICAL DATA VIEW: CUSTITEM *                      PAGE:  1
                              ******************************

LOGICAL FILE: DBFILE0   FILE DEFINITION: CUSTOMER                                DDNAME: CUSTFILE
----------------------
LOGICAL FILE: DBFILE1   FILE DEFINITION: ITEM    SYNCHRONIZING FIELD:  0.ITEMORD  DDNAME: ITEMFILE
----------------------
                          ------------ FIELD ALIASES-----------
                        ITMPRICE IS RENAMED TO ITEMPR
                        QTYBKORD IS RENAMED TO QTYBACKO
                        CUSTNO   IS RENAMED TO CUSTNBR
                        ORDERNO  IS RENAMED TO ORDERNBR
```

Figure 5-3 Logical Data View Formatted Source Listing

Synchronizing Files in a Logical Data View

You can synchronize any file in a logical data view with any other file.

- The synchronization process is driven from the primary file.
- Records are retrieved if they have key values equal to the synchronizing field.

Chained Synchronization

Figure 5-4 shows this type of synchronization, referred to as chained synchronization.

In the example, FILE1 is synchronized with the PRIMARY file when the X fields match; FILE2 is synchronized with FILE1 when the Y fields match.

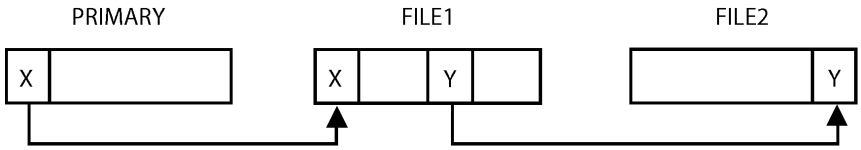

Figure 5-4 Chained Synchronization

Keyed files are synchronized only when their record access fields match.

- The synchronization is automatic and depends only on the contents of the specified synchronizing field.
- The synchronized file is accessed automatically the first time you reference a field from the synchronized file.
- Synchronization also occurs when the value of a synchronization field changes.
- A file is synchronized based upon the contents of its synchronization field.
  - If this logical field changes values between references to fields on the synchronized file, the file is resynchronized.
  - If it does not change, the file is not resynchronized.

Aliases in a Logical Data View

The purpose of a logical data view is to join different database managers. This is achieved by specifying the synchronization rules between the logical records that are derived from each of the databases.

You can define the logical records by the file definition process (see Chapter 4, *Understanding Files*). However, since each of the logical records are defined independently, it is possible that there may be a duplication between the names of the fields as they are defined for the different files. Since the end user accesses information from the database merely by using the field names, the duplication of names would cause ambiguity and result in incorrect information being returned. VISION:Inform uses the concept of alias names to resolve duplicate names from different file definitions.

Using Alias Field Names

Using alias names, VISION:Inform keeps the field names unique across all file definitions. VISION:Inform helps you resolve any ambiguity by providing the capability to create alias names for fields within the logical data view.

- Thus, you can resolve any conflict of names simply by aliasing one of the fields within the logical data view.
- The alias applies only to the logical data view for which it is defined. It has no effect on the original file definition of the logical record.

You can provide aliases while defining a logical data view. You can also provide alias names for segments if duplication occurs on the segment name.

Dynamic and Static Optimization

VISION:Inform automatically reads records from physical databases for the purpose of constructing logical records in computer memory. Different database managers have different methods of optimizing the access of data.

- DB2 uses the WHERE clause to optimize access to rows of a table, whereas, IMS uses the SSA (Segment Search Argument) for the purposes of optimizing access to segments of an IMS file.
- By default, VISION:Inform automatically constructs WHERE clauses for DB2 and SSAs for IMS depending upon the query specifications of the end users.

Batching and Constructing WHERE Clauses

The VISION:Inform Background Processor has the capability of batching together many different requests from different users to access data from the database. Thus, VISION:Inform passes the database only once for the purpose of extracting data for several users. VISION:Inform analyzes the different requests for data from all users batched together, and based upon these requests constructs suitable WHERE clauses or SSAs, which result in optimal performance.

Dynamic Optimization

Since the different requests batched together from different users vary from run to run, this analysis is performed each time prior to passing the database. This form of optimization is called dynamic optimization.

Providing Your Own WHERE Clauses

However, there are times when you know the characteristics your database better than VISION:Inform can deduce from the nature of the queries provided. Thus, VISION:Inform provides you with the capability of specifying your own WHERE clauses with DB2 or generate your own SSAs for IMS. You can specify WHERE clauses and SSAs to VISION:Inform within the logical data views in different ways.

- When you define a logical record generated from a DB2 database, you can display a panel on which you can provide the necessary WHERE statements.
- You can display this panel when you define the synchronization rules between the DB2 logical record and any other logical record.

In order to generate your own SSAs for IMS, you must provide special logical data view procedures of type P. These are procedures containing logical expressions which are compiled into SSAs. For information about the nature of these procedures, see Optimizing IMS Database Access in Chapter 4, *Understanding Files*, .

Static Optimization

When you provide either WHERE clauses or Preselection procedures for SSAs, you are defining the optimization to be used within the logical data view. This form of optimization is known as static optimization.

Disabling Dynamic Optimization

Static optimization and dynamic optimization are mutually exclusive. Therefore, in order to be able to use static optimization, you must disable dynamic optimization. The process of disabling dynamic optimization is described in the *Advantage VISION:Inform Installation Guide* for your environment.

Primary Names and Alternate Names

When you define fields in logical records, you can provide a primary name which is up to eight characters in length and an alternate name which can be up to thirty characters in length.

The alternate name capability is provided so that end users, who are accustomed to DB2 column names or to longer COBOL names, can continue to use them when requesting data from the client systems.

Using the Primary Name in the Logical Data View

However, when using field names anywhere within the logical data view, you must use the primary name.

- When you specify the fields which will synchronize the logical records, you must use the primary names.
- If you reference any fields from a logical record within your WHERE statement you must use the primary names. The names that you use within your Preselection procedures must also be the primary names.

Procedures in the Logical Data View (LDV)

The logical data view definition provides two powerful features:

- The first defines the joining of logical records from different physical databases, such as the synchronization of logical records.
- The second powerful capability is that of providing 4GL procedures for the purpose of data transformation or manipulation.

The steps that VISION:Inform executes in the process of honoring the client platform system request for data are as follows:

1. Synchronize the logical records according to the rules of the LDV.
2. Execute all LDV procedures against the logical records.
3. Execute all of the end user or client system's request for data.

Thus, the client system only has access to the data after it has been synchronized and modified by the logical data view procedures.

All procedures are written in a comprehensive powerful 4GL known as ASL. The syntax of ASL is discussed in *Advantage VISION:Inform ASL (Advanced Syntax Language) Reference Guide*.

When writing procedures within the logical data view, you can only use the primary name from any of the logical files. See Chapter 7, *Understanding Procedures* for information about procedures.

Types of Procedures

There are three types of procedures: N, S, and P.

Type N Procedures

Type N procedures can be considered to be the highest level of procedure. These procedures are main line procedures. Type N procedures are executed in the order of appearance in the logical data view.

Type S Procedures

Type S procedures are subroutine procedures. These procedures are activated only when called from another procedure. Type S procedures can be called from several different procedures.

Type P Procedures

Type P procedures are special kinds of procedures which optimize IMS calls by preselecting the data. These are not executable procedures but procedures which are compiled into an SSA for the purpose of accessing IMS database segments. See Dynamic and Static Optimization for information about preselection procedures.

Memory Optimization

VISION:Inform normally reads an entire database record into storage prior to processing any query, resulting in the following advantages:

- This method saves disk access at the expense of memory usage.
- This is generally the best approach for processing batches of queries.

Using this method, multiple data extractions to satisfy the multiple queries in each batch can be done during a single pass through the databases.

There are times, however, when the physical record described in the file definition or logical data view is too large to reside in memory. VISION:Inform provides you with a method with which you can still process queries against large database records.

When defining or updating a logical data view, you are provided with the opportunity to specify memory optimization for the logical data view.

Specifying memory optimization causes VISION:Inform to access and act on only one occurrence of each database segment in storage at any given point in time.

Performance Trade-offs

Memory optimization dramatically diminishes the amount of memory required to hold the physical database record. The benefit of decreased memory usage however, causes an increase of disk access.

The disk access increase caused by memory optimization can, in some cases, be dramatic, so use this feature only when necessary.

Chapter

6

Understanding Tables

A table is an orderly arrangement of data in rows and columns where each item (cell) can be unambiguously identified by one or more arguments. A table is also an array of data in which each item is uniquely identified by a label, by its position relative to other items, or by some means.

Table Concepts

VISION:Inform provides the capability to use encoded data within the database to save space within the records, while returning the data to the user in an expanded form.

For example, it is often convenient to store department codes, such as 105, but when the data is retrieved, to view it as "ACCOUNTING."'

This process is known as table lookup. The codes are stored as the argument of a table while the expanded description is stored as the result of a table.

The following illustrates a typical table:

Argument	Result
101	ENGINEERING
102	SALES
103	MARKETING
104	QUALITY CONTROL
105	ACCOUNTING

Figure 6-1 The DEPARTMENT Table

The table lookup process is fully automatic to the client system within VISION:Inform.

Defining the Table Lookup Process

The database administrator defines the lookup process either by using the automatic lookup feature provided within the file definition (see Chapter 4, *Understanding Files*) or by using the ASL procedural language and performing the lookup in procedures within the logical data view (see Chapter 5, *Understanding Logical Data Views*).

You can use tables:

- To reduce file size by storing codes instead of strings of frequently used text.
- To store information used frequently by many applications.

When you promote a table definition, a formatted source listing is produced. The listing itemizes the table characteristics, size, table values, and associated argument values.

Tables can consist of any number of entries. Each table entry contains two elements: an argument value and a result value**.**

- Table arguments and results can be of any field type.
- The result field and argument lengths are limited by the field type you select.

Using Table Lookup

When using table lookup, an input argument (or field) is referenced. The input argument is compared to the table argument and when a match is identified, the table result is returned. This all happens automatically so that it appears as if you were simply processing the result field.

For example, assume we had a field in a record named DEPT, and this field were defined to be an automatic lookup field using the table DEPARTMENT in Figure 6-1. If the actual value of the DEPT field in the file is 104, a request to display the contents of the field DEPT would display the value QUALITY CONTROL. In other words, the user would not see the code 104, but would see the text result that corresponds to the code

Figure 6-2 shows a sample table formatted source listing.

```
                                    ******************************
09/22/01                            * TABLE LISTING FOR: ACCOUNT  *                  PAGE 1
                                    ******************************

                      LAST UPDATED BY:                    EXPIRATION DATE:   /  /
                                 TABLE TYPE: B    NUMBER OF TABLE ENTRIES:    4
ARGUMENT( TYPE: C  LENGTH:  1  DECIMAL PLACES:  )    RESULT( TYPE: C  LENGTH:  15  DECIMAL PLACES:)
-------------------------------------------------------------------------------------------------
                ARGUMENT                                                          RESULT
-------------------------------------------------------------------------------------------------
 0                                                  REVENUE
 5                                                  DIRECT
 6                                                  INDIRECT
 7                                                  G & A
```

Figure 6-2 A Table Formatted Source Listing

Table Types

The order in which comparisons between the input argument and table arguments are made depends on the type of table. There are three table types available:

- Displacement
- Sequential
- Binary

Binary tables are the most commonly used table type. The following sections explain each table type.

Searching Techniques

Once a search is started on a sequential or binary table, the input argument is compared with the table arguments until the comparison meets the conditions set for a successful search.

- When the search is successful, the result is returned to the result field.
- If the entire table is searched without a successful comparison, the results can vary depending on the table type and the type of comparison.
- "Not found" conditions are described in more detail later in this chapter.

Displacement Tables

Table searches of displacement tables operate differently from those of binary and sequential tables.

- The result values in displacement tables are stored in a specific order within the table.
- The table arguments are based upon the relative position of the result values from the start of the table.
- When you request a result from a displacement table, a search is not performed.
- The result is retrieved directly by its position in the table.
- The position within the table is computed from the value of the input argument.

Figure 6-3 shows a displacement table containing the months of the year. The table arguments are the integers 1 through 12. In this example, if the input argument is 9, the result value SEPTEMBER is retrieved directly from the table.

Argument (Position)	Result
1	JANUARY
2	FEBRUARY
3	MARCH
4	APRIL
5	MAY
6	JUNE
7	JULY
8	AUGUST
9	SEPTEMBER
10	OCTOBER
11	NOVEMBER
12	DECEMBER

Figure 6-3 A Displacement Table

Displacement Table Arguments

Table arguments for displacement tables are always contiguous integers from 1 to the largest argument value in the table.

- If you define a displacement table in which the table arguments are not contiguous, the table is stored with dummy entries for the missing arguments.
- Missing arguments are marked internally as unused.

Displacement Table Considerations

Displacement tables are searched more quickly than sequential or binary tables, but displacement tables are most appropriate for highly-structured data such as the example shown in Figure 6-3. If your table arguments are not integers or there are many gaps in the sequence, you should consider one of the other table types.

Displacement Table Characteristics

When you define a displacement table, you can specify the entries in any order.

- The entries are sorted before they are stored in the library.
- At any time, you can add new entries to the end of the table or replace existing, dummy, or deleted values.
- The argument is not physically stored in the table.

Sequential Tables

The values stored in sequential tables are stored in an unspecified order. In a sequential table, the search begins with the first argument in the table and proceeds, one argument at a time, until a table argument matches the input argument.

You can use any type of table arguments, and the arguments can be in any order.

- Because the search is made through every argument until a match is found, you should arrange the table with the most frequently referenced table arguments at the beginning of the table.
- This minimizes the average search time and makes this type of table particularly useful when a small number of arguments are frequently referenced and the remainder are rarely used.

New arguments are always added to the end of a sequential table. The table is not ordered when it is stored in the library. The table is subsequently used in the order in which you specify the entries. You can add or delete table arguments without affecting any other arguments.

Figure 6-4 shows a sequential search table relating ZIP codes to city names.

A search against this table begins with the first table argument, 91311, and continues through all the table arguments until a match with the input argument is found.

Using 91344 as the input argument, the result, GRANADA HILLS, is returned after five search comparisons.

For this example table to be efficient, the input arguments used must correspond to the first few table arguments (such as 91311, 91303, and 91316).

Argument	Result
91311	CHATSWORTH
91303	CALABASAS
91316	ENCINO
91331	PACOIMA
91344	GRANADA HILLS
91356	TARZANA
91024	NORTHRIDGE
91345	MISSION HILLS
91342	SYLMAR
91306	CANOGA PARK
91343	SEPULVEDA
91401	VAN NUYS
91335	RESEDA
.	
.	
.	

Figure 6-4 A Sequential Table

Binary Tables

Binary tables have entries ordered according to argument values.

When you search this type of table, the input argument is first compared with the table argument at the midpoint of the table.

- If the input and table arguments meet the comparison criteria, the search is successful and the result is returned in the result field.
- If the input argument is less than the table argument, it is next compared with the midpoint of the first half of the table. Likewise, if it is greater, it is next compared with the midpoint of the last half of the table.

This comparison with the midpoints of decreasingly smaller sections of the table continues until the comparison criteria are met or the table cannot be further divided.

Figure 6-5 shows a search of a binary table.

Number of Searches	Argument	Result
	001	JOHN COATES
	025	FAYE HALPER
	122	ROGER KIMBALL
	240	CAROL SINGER
1 --->	310	PHILLIP KEYS
3 --->	343	DONALD SULLIVAN
2 --->	360	IRVING MALLMAN
	382	RONALD GREEN
	416	GEORGE JONES

Figure 6-5 A Search of a Binary Table

The table contains employee numbers as table arguments and names as results.

- If you want to find the name of the employee whose number equals 343, the input argument 343 is compared to the midpoint table argument 310.
- Since the input argument is greater than the midpoint, 343 is next compared to the midpoint of the last half of the table, 360.
- This time the input argument is less then the midpoint, so the input argument is again compared to the midpoint where a sole entry of the next section is 343.
- The arguments match, so the search is successful and DONALD SULLIVAN is returned as the result after only three comparisons.

Binary Table Advantages

Binary search tables are the most commonly used table type. They are versatile in that:

- You can use table arguments of any type.
- Entries can be specified in any sequence and are ordered when stored in the library.
- Large amounts of table entries can be created.
- Storage is used for only those entries that exist physically in the table.
- Binary tables are the only type of table that permits comparisons other than EQUAL (such as NEAREST value, NEXT SMALLER value, NEXT GREATER value, and INTERPOLATION).

If the arguments do not meet the special purposes of the other table types, binary search is the most efficient.

Number of Comparisons for Each Table Type

Figure 6-6 shows the maximum number of comparisons that can be made with each table type.

Number of Table Entries	Maximum Number of Comparisons		
	Binary	Sequential	Displacement
10	4	10	1
100	6	100	1
500	9	500	1
1000	10	1000	1
8000	13	8000	1

Figure 6-6 Maximum Number of Comparisons by Table Type

Types of Comparisons for Binary Tables

A comparison between an input argument and table argument is successful depending on the conditions set when you request the search. When the comparison is successful, a result value is returned to the result field. Otherwise, the result field becomes invalid. There are five types of comparisons for binary tables:

- Return the result of the table argument that is EQUAL to the input argument.
- Return the result of the table argument that is NEAREST to the input argument.
- Return the result of the table argument that is NEXT SMALLER or EQUAL to the input argument.
- Return the result of the table argument that is NEXT GREATER or EQUAL to the input argument.
- Return a result INTERPOLATION from the results of the two table arguments between which the input argument falls.

Displacement and sequential tables permit only EQUAL comparisons. The NEAREST, NEXT SMALLER, NEXT GREATER, and INTERPOLATION comparisons are only available with binary tables. See Binary Tables: Comparing for an EQUAL Condition for table examples that solve the same problem using the different comparison methods.

Binary Tables: Comparing for an EQUAL Condition

The most common type of comparison in binary tables is the EQUAL condition. You can use EQUAL comparisons with any table type and with any type of table arguments.

Figure 6-7 shows a table search that uses an EQUAL comparison to convert the month to a quarter of the year. The table contains 2-digit month designations for the table argument and a number representing the quarter for the result.

If you want to find the quarter for the sixth month, you can specify an EQUAL search with an input argument of 06. When the search reaches the 06 table argument, you get a result of 2.

Argument Month	Result Quarter
01	1
02	1
03	1
04	2
05	2
06	2
07	3
08	3
09	3
10	4
11	4
12	4

Figure 6-7 A Conversion Table for an EQUAL Comparison

Binary Tables: Comparing for the NEAREST Value

For a NEAREST comparison, the table arguments must be numeric.

Figure 6-8 shows a NEAREST comparison where the table argument nearest (or equal) to the input argument in value satisfies the condition. The input argument, 06, is between the table arguments 05 and 08. Because 06 is nearer in value to 05, the result value 2 is returned.

If the input argument is midway between two table arguments, the result of the greater argument is returned. For example, the input argument of 3 uses the table argument of 5 and returns a result of 2.

Argument Month	Result Quarter
01	1
05	2
08	3
11	4

Figure 6-8 A Conversion Table for a NEAREST Comparison

Binary Tables: Comparing for the NEXT SMALLER (or EQUAL) Value

Figure 6-9 shows a NEXT SMALLER comparison where the table argument that is next smaller than (or equal to) the input argument satisfies the condition. The input argument, 06, is between the table arguments 04 and 07. The argument is equal to neither, but 04 is the next smaller of the two, so the result returned is 2.

Argument Month	Result Quarter
01	1
04	2
07	3
10	4

Figure 6-9 A Conversion Table for a NEXT SMALLER (or EQUAL) Comparison

Binary Tables: Comparing for the NEXT GREATER (or EQUAL) Value

Figure 6-10 shows a NEXT GREATER comparison where the table argument that is next greater than (or equal to) the input argument satisfies the condition. The input argument, 07, is between the table arguments 06 and 09. The argument is equal to neither, but 09 is the next greater of the two, so the result returned is 3.

Argument Month	**Result Quarter**
03	1
06	2
09	3
12	4

Figure 6-10 A Conversion Table for a NEXT GREATER (or EQUAL) Comparison

Binary Tables: INTERPOLATION Comparisons

An INTERPOLATION comparison returns a result linearly interpolated from the results of the table arguments between which the input argument falls. The equation used is:

$$\text{Result} = (\text{Input} - \text{ArgumentL}) \frac{\text{ResultH} - \text{ResultL}}{\text{ArgumentH} - \text{ArgumentL}} + \text{ResultL}$$

Where :

Input	= input argument value.
ArgumentL	= table argument less than the input argument.
ArgumentH	= table argument greater than the input argument.
ResultL	= result value of ArgumentL.
ResultH	= result value of ArgumentH.

Because interpolation requires numeric operations, both the table arguments and the results in interpolated tables must be numeric.

Figure 6-11 shows part of a tax table for taxpayers with incomes from $10,000 to $60,000. Figure 6-12 shows the same values stored in a binary table for interpolation.

To calculate the tax for someone earning $23,000 from the table, find the portion of the table including this value. You find $23,000 in a bracket from $22,000 to $26,000. Tax on $23,000 is calculated by adding 40% of any earnings over $22,000 to the tax of $5,990. In this case, the total tax is $6,390.

Income		**Tax to Pay**	
Over---	But Not Over---	Tax---	of Excess Over---
$10,000	$12,000	$ 2,090+27%	$10,000
$12,000	$14,000	$ 2,630+29%	$12,000
$14,000	$16,000	$ 3,210+31%	$14,000
$16,000	$18,000	$ 3,830+34%	$16,000
$18,000	$20,000	$ 4,510+36%	$18,000
$20,000	$22,000	$ 5,230+38%	$20,000
$22,000	$26,000	$ 5,990+40%	$22,000
$26,000	$32,000	$ 7,590+45%	$26,000
$32,000	$38,000	$10,290+50%	$32,000
$38,000	$44,000	$13,290+55%	$38,000
$44,000	$50,000	$16,590+60%	$44,000
$50,000	$60,000	$20,190+62%	$50,000

Figure 6-11 Tax Table for Taxpayers

To find the same tax interpolating from the binary table, the table is searched for table arguments bracketing $23,000. The input argument falls between $22,000 and $26,000.

Using the equation given above:

$$\text{Result} = (\$23{,}000\text{-}\$22{,}000)\,\frac{\$7{,}590\text{-}\$5{,}990}{\$26{,}000\text{-}\$22{,}000} + \$5{,}990$$

Result = $6,390

The result calculated is half-adjusted by adding five to the rightmost (low-order) digit, then truncating that digit. If the result value is negative, five is subtracted from the rightmost digit.

Argument Income	Result Tax
10,000	$ 2,090
12,000	$ 2,630
14,000	$ 3,210
16,000	$ 3,830
18,000	$ 4,510
20,000	$ 5,230
22,000	$ 5,990
26,000	$ 7,590
32,000	$10,290
38,000	$13,290
44,000	$16,590
50,000	$20,190

Figure 6-12 A Conversion Table for INTERPOLATION

Argument Not Found Conditions

If the comparison of arguments causes a search outside of the range of the table, the result value is set to the value shown in Figure 6-13.

Comparison Condition	Input Argument Value	
	Above Table Range	Below Table Range
EQUAL	(Invalid)	(Invalid)
NEAREST	Highest Entry	Lowest Entry
NEXT SMALLER	Highest Entry	(Invalid)
NEXT GREATER	(Invalid)	Lowest Entry
INTERPOLATION	(Invalid)	(Invalid)

Figure 6-13 Search Results for Out-of-Range Input Arguments

Invalid data displays as a special symbol (*, -, or +) when the data is output.

Automatic Table Lookup

You can define table lookup when you define the file (see Chapter 4, *Understanding Files*). This form of table lookup is called automatic table lookup.

Table searches are transparent with automatic table lookup. Any reference that you make to a table field, results in a table lookup.

- If the table lookup is successful, the appropriate table value is returned.
- If the lookup is not successful, the field is marked as invalid.

You can use automatic table lookup with any table type and with any comparison method allowed by the table type.

Defining a Result Field

The field you reference that starts the table search is called a result field. It contains the result of the table search.

You define a result field in a file definition the same way you define any other file field.

- In most respects, result fields are similar to other fields with the exception that they do not physically exist in the file.
- When you define result fields, you do not define field attributes; you specify the result field attributes in the table definition.
- For a result field, you specify the table you want searched, which field to use as the input argument, and the comparison method.
- The file definition output source listing shows result field names along with regular file fields.

Using Input Arguments

For an input argument, you can use any field in the file that is not a result field.

- Input arguments used to search sequential and binary tables do not require the same attributes as the table argument. If the attributes do not agree, the input argument attributes are automatically converted to those of the table argument.
- Input arguments used with displacement tables can be of any data type, but only the integer portion is used during the search.

Your file definitions can contain any number of result fields. Different result fields can refer to the same table and use different input arguments. They can also use the same input argument and refer to different tables.

Modifying Result Fields

If you modify a table so that its result field characteristics are changed, remember to modify any automatic table lookup against that table.

You can delete, replace, and create result field definitions in a file definition in the same way as other fields. See the *Advantage VISION:Inform Definition Processor Reference Guide* for more information about the specifications.

Procedural Table Lookup

You can also include table lookup commands in procedures that you use within a logical data view (see Chapter 5, *Understanding Logical Data Views*). The LOOKUP function in ASL, the language you use to write procedures, performs the table lookup.

The following ASL statement illustrates a typical table lookup:

```
LET T.MONTH = LOOKUP (MONTHS, MM)
```

This statement takes the value in the input argument MM, performs a table lookup on the table MONTHS and returns the result to the field T.MONTH.

Chapter

7

Understanding Procedures

A procedure is a set of statements that perform a specific function.

Procedure Concepts

You write procedures using Advanced Syntax Language (ASL), a powerful fourth generation language (4GL), supported by VISION:Inform. For information on using ASL, see the *Advantage VISION:Inform ASL (Advanced Syntax Language) Reference Guide*.

Procedures are useful for performing data transformations or including business algorithms necessary for the extraction of data from databases.

There are two types of procedures:

- Logical data view procedures (LDV procedures).
- Generalized Data Base Interface (GDBI) mapping procedures.

You write both of these types of procedures using ASL. See Logical Data View Procedures and Generalized Data Base Interface (GDBI) Procedures for information about the differences between these types of procedures.

Logical Data View Procedures

Logical data view procedures are procedures which have access to all of the data that is extracted from the database. They can manipulate the data, make calculations, and generally apply business algorithms as appropriate.

Logical data view procedures are procedures, written in ASL, which are included within the logical data view. The action of the logical data view is first to synchronize the logical records constructed from the different databases and then to apply the logical data view procedures against the synchronized and extracted data.

Use logical data view procedures:

- To transform data before it is sent to the users (for example, to convert inches to centimeters).
- To provide data security, where the procedures check that the relevant data should be passed on to the end users.
- To dynamically change the synchronization of the logical records.

  Logical records are synchronized by matching the key field of the logical record against a field from another file. Normally, this amounts to synchronizing key fields between logical records. However, the synchronizing field may also be a temporary field which is computed dynamically within a procedure. Thus, a procedure can control the records that are to be synchronized.

Logical data view procedures and GDBI mapping procedures operate independently. They do not interfere with each other.

Generalized Data Base Interface (GDBI) Procedures

Note: See *Chapter 9, GDBI Processing* for information about GDBI.

Using ASL, you write GDBI mapping procedures to build logical records. VISION:Inform automatically builds logical records from IMS, VSAM, and DB2 databases. You use GDBI mapping procedures to build logical records from non-IBM databases, such as ADABAS and IDMS.

You can also use GDBI mapping procedures with these databases when you need to build a logical record that has certain characteristics not readily definable using the VISION:Inform Definition Processor.

Conceptually, you can think of GDBI mapping procedures as substituting for the calls to the database manager generated by VISION:Inform. In effect, when VISION:Inform is about to make a call to the database manager for the purpose of

retrieving fields from the database, if there are GDBI mapping procedures in existence, VISION:Inform transfers control to these procedures and leaves these procedures with the responsibility for retrieving the data.

GDBI mapping procedures may co-exist with logical data view procedures. The different procedure types do not interfere with each other.

Writing Procedures

VISION:Inform provides a virtual relational table view of data for the end user. As far as the end user is concerned, it is as if the user had a single relational table. Each of the columns of the table has a name. The end user is not concerned with the physical databases from which the data is retrieved.

However, the procedures used within VISION:Inform are part of the process of building the virtual relational table. Because of this, a procedure must know the location of the record in order to access the data.

Using Alias Names

Since logical records are defined at different times, there is the possibility that the same name is used in different logical records. This ambiguity is resolved for the end user by providing an alias naming capability within the logical data view. (See Aliases in a Logical Data View in Chapter 5, *Understanding Logical Data Views* for more information.) Thus, the end user is provided with a distinct set of names.

Procedures work at a much lower level and they do not have access to these alias names, so the procedures must identify the fields within the logical records by a qualifier or tag, which precedes the field name. Thus, field names from DBFILE0 and DBFILE1 are distinguished as follows:

0.SALARY (from DBFILE0)

1.SALARY (from DBFILE1)

The list of qualifiers is shown in Figure 7-1.

Qualifier	Logical Source
0	Primary File (DBFILE0)
1	Synchronized File 1 (DBFILE1)
2	Synchronized File 2 (DBFILE2)
3	Synchronized File 3 (DBFILE2)
4	Synchronized File 4 (DBFILE4)
5	Synchronized File 5 (DBFILE5)
6	Synchronized File 6 (DBFILE6)
7	Synchronized File 7 (DBFILE7)
8	Synchronized File 8 (DBFILE8)
9	Synchronized File 9 (DBFILE9)
F	System Field
T	Temporary Field

Figure 7-1 Qualifiers Used by VISION:Inform Procedures

Fields that are defined by means of the Definition Processor can have a primary name (1 to 8 characters) and an alternate name (1 to 30 characters). Use only the primary names in procedures.

Types of Procedures

There are three types of procedures: N, S, and P.

Type N Procedures

Type N procedures are the highest level of procedure. These procedures are main line procedures. Type N procedures are executed in the order that they appear in the logical data view.

Type S Procedures

Type S procedures are subroutine procedures. These procedures are only activated when called from a another procedure. Type S procedures can be called from several different procedures.

Type P Procedures

Type P procedures are special kinds of procedures which optimize IMS calls by preselecting the data. These are not executable procedures but procedures which are compiled into an SSA for the purpose of accessing IMS database segments. See Optimizing IMS Database Access and Dynamic and Static Optimization in Chapter 4, *Understanding Files* for more information about preselection procedures.

System Fields

There are many pieces of information available within the VISION:Inform system that are useful to control the logic of the procedures. This information is provided to the procedures in a series of fields which are called system fields. System fields have primary names just like any other fields, and they are identified by prefixing with the qualifier F (such as F.DATE).

Because logical data view procedures and GDBI mapping procedures effectively operate in different environments, they have different demands on the information that is provided by VISION:Inform, and in some cases, returned to VISION:Inform. Thus, the procedures have access to different sets of system fields. Figure 7-2 shows an alphabetical the list of system fields that are available to the procedures.

Note: Y = AVAILABLE and N = NOT AVAILABLE.

Explanations of system fields follow the table.

System Field	Logical Data View	GDBI
CHKP	Y	Y
CKPTID	Y	Y
COMMAND	N	Y
CONDCODE	Y	N
CSTATUS	Y	Y
DATE	Y	Y
ECORD	Y	N
EOF	Y	N
FDNAME	N	Y
FILE	N	Y
FILEID	N	Y
ISDATE	Y	Y
JULANX	Y	Y
JULIAN	Y	Y
LILIAN	Y	Y
LNUMBER	Y	Y
LSTART	Y	Y
MNUMBER	Y	Y
MSTART	Y	Y
MSTATUS	N	Y
MODE	N	Y
PASSWORD	N	Y
RESTART	Y	Y
RETURNCD	Y	Y
RNUMBER	Y	Y
RSTART	Y	Y
SEGNAME	N	Y
SSCOUNT	Y	Y
TIME	Y	Y
TODAY	Y	Y
TODAYX	Y	Y

Figure 7-2 System Fields Available to Procedures

CHKP (IMS Systems Only)

The CHKP system field triggers a checkpoint operation by storing a non-blank value into the system field from any procedure except preselection (type P) procedures. The checkpoint does not occur until just before VISION:Inform is ready to read the next root segment. The system field is reset to a blank after every checkpoint. The user can enter an A in the CHKP system field which forces an ABEND to occur rather than a checkpoint. The ABEND code issued will be 117.

CKPTID (IMS Systems Only)

The CKPTID system field contains the ID of the last checkpoint taken. It contains blanks prior to the first checkpoint.

COMMAND

An 8-character field which contains the command identifier. You use the COMMAND system field with GDBI procedures. Command values are:

Sequential or Serial:

GETFIRST	Get first segment within a parent.
GETNEXT	Get the next occurrence of the segment type previously obtained.

Key Driven:

GETFKEY	Get first segment with key greater than or equal to the supplied value.
GETKEY	Get the segment with the key provided.
INIT	Initialization procedure call.
TERM	Termination procedure call.

The information is primarily useful for directing the operation of the GDBI mapping procedure, but can also be useful to a database manager. The system fields are initialized prior to entering a GDBI mapping procedure.

CONDCODE

The CONDCODE system field provides communication with your job control.

At the end of the run, after all data extraction processing is completed and all files have reached end-of-file, the values you placed in the CONDCODE system field in your procedures are added to the condition code values normally supplied to the operating system by VISION:Inform.

Use this procedure to control the range of condition codes without losing the settings provided by VISION:Inform. Only the last value of CONDCODE is used; intermediate values have no effect. If the CONDCODE is invalid at the end of the run, VISION:Inform sets it to 20 before adding it to the normal condition code value.

It is your responsibility to ensure that the final value of the condition code is recognized by the operating system.

CSTATUS

The call status system field (CSTATUS) is a 4-byte character string flag that indicates the status of a call to an external subroutine. CSTATUS values are set by VISION:Inform and indicate whether a call was suppressed by VISION:Inform and why.

- A call is suppressed if a parameter specifies an invalid or missing field. The following table shows the settings of the CSTATUS system field.

Value	Meaning
PMIS	Parameter missing
PINV	Parameter invalid
Blank	Call successful

- The CSTATUS system field should be examined before using a value from the RETURNCD system field since a suppressed call would not allow the called routine to set the RETURNCD system field.

DATE

The DATE system field records the date in a 12-character format: MMM DD, YYYY, where MMM=3 alpha characters for month, DD=2 numerals for day of month, and YYYY=4 numerals for year.

ECORD

The ECORD system field tests the status of the synchronized files in relation to the primary file or to the synchronized file to which it is chained for equal (or match), high, or low conditions.

Ecord Flag Values:

Value	Meaning
M	The synchronized file and primary file keys are equal. The synchronized file has a match, and its record is available for processing.

X	The synchronized file record is not available for processing. This value is also used when the synchronized file reaches the end of file or when no synchronized files exist for the run.

In the ECORD system field, each character indicates the current status of a corresponding synchronized file. Partial fielding is used to determine the status of one or more of the synchronized files, as shown in the following table.

Partial Field Specification and the ECORD Flag:

File Name	Partial Field Start and Length
DBFILE1	1,1
DBFILE2	2,1
DBFILE3	3,1
DBFILE4	4,1
DBFILE5	5,1
DBFILE6	6,1
DBFILE7	7,1
DBFILE8	8,1
DBFILE9	9,1

EOF

The EOF system field detects or forces an end of file on sequentially read input files. It is a character string field with a length of eleven bytes. It can be tested to determine if an input file exists. Alternatively, it can be used to terminate the reading of an input file by storing the appropriate value into the flag.

When forcing end of file, if any value except E is placed into the EOF system field, the results are unpredictable.

Each character pertains to an input file as shown in the following table, and can have one of three values:

Value	Meaning
E	The file has reached end of file.
Y	The file has not reached end of file.
N	The file does not exist.

Input File Name	Partial Field Specification
DBFILE0	1,1
DBFILE1	3,1
DBFILE2	4,1
DBFILE3	5,1
DBFILE4	6,1
DBFILE5	7,1
DBFILE6	8,1
DBFILE7	9,1
DBFILE8	10,1
DBFILE9	11,1

FDNAME

An 8-character field which contains the file name.

It is initialized prior to entering a mapping procedure. The information in this system field is generally required by database managers.

FILE

An 8-character field which contains the DD name as provided to the Definition Processor.

The information is primarily useful for directing the operation of the mapping request, but can also be useful to a database manager. It is initialized prior to entering a mapping request.

FILEID

An 8-character field which contains the file identification.

It is initialized prior to entering a mapping procedure. The information in the field is generally required by database managers.

ISDATE

This system field records the date in an international standards organization date format of YYYYMMDD, where YYYY=4 numerals for year, MM=2 numerals for month and DD=2 numerals for day (for example, April 14, 2001=20010414).

JULANX

System date recorded in YYYYDDD format, where YYYY=year and DDD=day in the year (for example, April 14, 2001=2001105)

JULIAN

This system field records the date in the format of YYDDD, where YY=year and DDD=day in the year (for example, April 14, 1998=98105).

LILIAN

4-byte binary representation of date, using an informal format.

LNUMBER

The LNUMBER system field is a 4-byte fixed-point text processing partial field system field that specifies the number of characters in the left part of the field.

- You can use it in a scan left or right operation.
- You can test it, modify it, or use it in computation during procedure execution.
- You can use the value (LN) in the partial field specification in order to access a particular area of the scanned field.
- Since the value of LNUMBER can change during processing, you can use this system field to perform dynamic partial field operations.

A dynamic partial field specification might be in error during processing because:

- The partial field flag fields are invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid.

LSTART

The LSTART system field is a 4-byte fixed-point text processing partial field flag that specifies the starting position of the left part of a field.

- You can use it in a scan left or right operation.
- You can test it, modify it, or use it in computation during request processing.
- You can use the value (LS) in the partial field specification in order to access a particular area of the scanned field.

A dynamic partial field specification might be in error during processing because:

- The partial field flags are invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid.

MNUMBER

The MNUMBER system field is a 4-byte fixed-point text processing partial field flag that specifies the number of characters in the middle (matching) part of a field.

- You can use it in a scan left or right operation.
- You can test it, modify it, or use it in computation during request processing.
- You can use the value (MN) in the partial field specification in order to access a particular area of the scanned field.

A dynamic partial field specification might be in error during processing because:

- The partial field system fields are invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid.

MODE

A 2-character system field which contains information about the application mode of operation.

SR	Standard Processing
MR	Storage Optimized Processing

It is initialized prior to entering a GDBI mapping procedure. The information in the system field is generally required by database managers.

MSTART

The MSTART system field is a 4-byte fixed-point text processing partial field flag that specifies the starting position of the middle (matching) part of the field.

- You can use It in a scan left or right operation.
- You can test it, modify it, or use it in computation during request processing.
- You can use the value (MS) in the partial field specification in order to access a particular area of the scanned field.

A dynamic partial field specification might be in error during processing because:

- The partial field flags are invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid.

MSTATUS

The MSTATUS system field is a 6-character field used by the GDBI mapping procedures to instruct VISION:Inform on a specific action to take subsequent to completion of the GDBI procedures. MSTATUS is initialized to blanks prior to each call to a GDBI procedure.

Permissible return values:

Value	Explanation
blank	Mapping successfully completed.
NFOUND	Segment not found. Signifies to VISION:Inform that there was no data to fill the skeleton segment either because of having run out of repeated segments or failure to locate a keyed segment. In the process of building a hierarchical record, VISION:Inform returns to the GDBI procedure for each segment type to request repeated segments. Thus, the GDBI procedure must be programmed to issue NFOUND for each segment type to terminate a series of repeated segments.
STOP*nn*	Stop the run and set the condition code to *nn*. Used by GDBI procedures to cease processing.

The information is primarily useful for directing the operation of the GDBI procedure, but can also be useful to a database manager.

PASSWORD

The PASSWORD system field contains the password as provided for the appropriate logical file. It is available for all procedures, but is expected to be most valuable to the initialization procedures to control access to the database. This field is set prior to a GDBI procedure receiving control.

RESTART (IMS System Only)

The RESTART system field is an 8-byte character field that determines the restart status of a VISION:Inform run. If a run has been restarted, the RESTART system field contains the checkpoint ID at which the restart occurred. Otherwise, the value of the RESTART system field is all blanks.

RETURNCD

The return code system field (RETURNCD) is a 4-byte fixed-point field that reflects the status of a call. The values are set by the called routine, and the contents can be examined after a call. When the system field is included in reports, VISION:Inform edits it as though it was a 2-byte fixed point field (output width of 7, maximum printable value 99,999). The value in this field is the value in general register 15 upon return to VISION:Inform from a called routine.

RNUMBER

The RNUMBER system field is a 4-byte fixed-point text processing partial field flag that specifies the number of characters in the right part of the field.

- You can use it in a scan left or right operation.
- You can test it, modify it, or use it in computation during request processing.
- You can use the value (RN) in the partial field specification in order to access a particular area of the scanned field.

A dynamic partial field specification might be in error during processing because:

- The partial field system fields are invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid.

RSTART

The RSTART system field is a 4-byte fixed-point text processing partial field system field that specifies the starting position of the right part of the field.

- You can use it in a scan left or right operation.
- You can test it, modify it, or use it in computation during request processing.
- You use the value (RS) in the partial field specification in order to access a particular area of the scanned field.

A dynamic partial field specification might be in error during processing because:

- The partial field system fields are invalid.
- The partial field specification is outside the current size of the field.
- The starting character or number of characters value is less than 1.

If the dynamic partial field specification is in error, the field to which it is applied becomes invalid for the current operation. If an invalid dynamic partial field is applied to a result field, the result field becomes invalid.

SEGNAME

An 8-character field which contains the segment name.

The information is primarily useful for directing the operation of the GDBI procedure, but can also be useful to a database manager. It is initialized prior to entering a GDBI procedure.

SSCOUNT

The SSCOUNT system field counts the number of matches found during the REPLACE. The system field is set to 0 at the start of each run.

Setting	Result
0	There were no successful substitutes.
-2	One of the operands of REPLACE is null or invalid.

TIME

The TIME system field records the time of day at which the run was started. The TIME system field prints as HH.MM.SS, where HH=hours, MM=minutes, and SS=seconds. Leading zeros are added where necessary.

TODAY

The TODAY system field records the date in the format MMDDYY, where MM=2 numerals for month, DD=2 numerals for day, and YY=2 numerals for year.

TODAYX

System date recorded in MMDDYYY format, where MM=2 numerals for month, DD=2 numerals for day, and YYYY=4 numerals for year.

Chapter

8

Processing Data Structures

VISION:Inform processes the data structures defined both automatically and under controlled conditions. See Chapter 4, *Understanding Files* for the definition of data structures.

The power of VISION:Inform lies in the method of processing data structures automatically. Once the essential concepts of automatic processing and controlled processing are clear, application development becomes easy.

Processing Structured Records

Suppose you have a structured file that can be represented by the hierarchical structure in Figure 8-1.

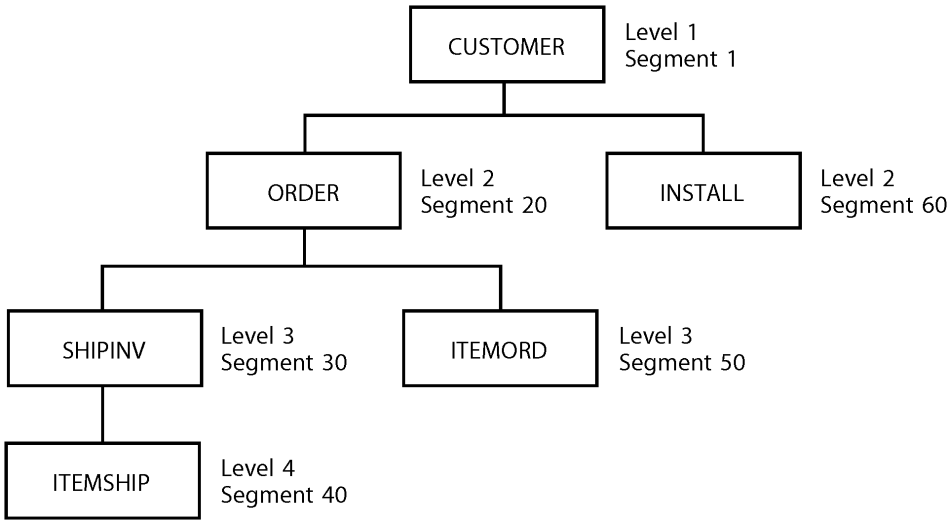

Figure 8-1 Structure of the CUSTOMER File

Now suppose you have customer A1 with three orders, B1, B2, and B3. A1 also has one installation, D1. Order B1 has two items, C1 and C2. Order B3 has one item, C3 (see Figure 8-2).

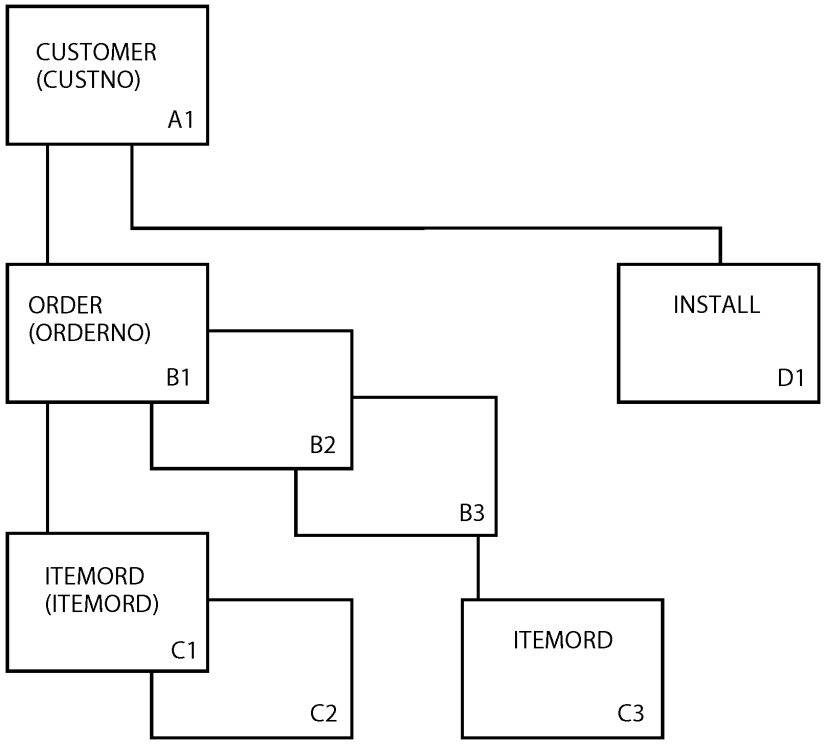

Figure 8-2 CUSTOMER A1 and Related Information

The CUSTOMER segment contains customer information such as name and phone. The ORDER segment contains order information for each customer. The ITEMORD segment contains item information for each customer.

Fields within the segments are shown in parentheses. The fields shown are referenced later in this chapter.

The next section describes how VISION:Inform processes structured records. The key concept is that of looping.

Looping

The first executed reference to any field in a lower level segment starts a loop on the occurrences of that segment type. The loop is automatically generated in VISION:Inform. Each occurrence for that segment type is automatically available for further processing.

For example, the fields selected in Figure 8-3 start a loop on the ORDER segment.

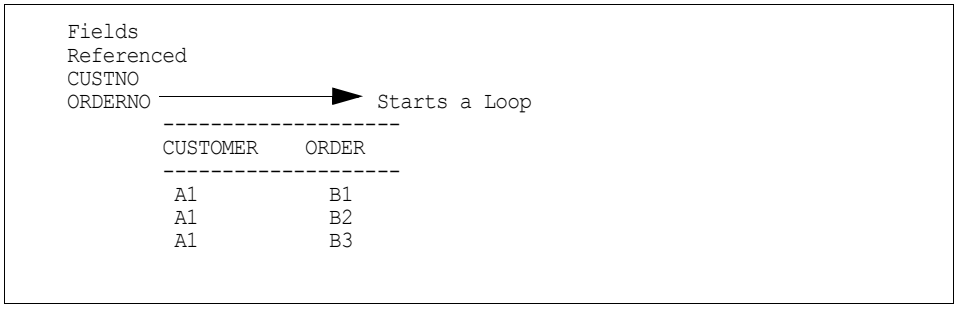

Figure 8-3 A Simple Loop

In this and subsequent illustrations, you list the fields referenced at the left of the illustration and the sets of data retrieved at the right of the illustration. It is assumed that the fields are referenced in the order of top to bottom of the listed fields.

Notice how the first reference to the ORDER segment starts a loop on all occurrences of ORDER for customer A1. Note how VISION:Inform outputs the records, repeating A1 for each lower level occurrence.

- **Simple Loops—**Simple loops are those which are initiated with reference to any lower level segment type.
- **Nested Loops**—A nested loop is one in which more than one lower level segment is referenced.

For example, in Figure 8-4, the statement that references ORDERNO starts the first loop; the report statement that references ITEMORD starts the second loop nested within the first loop. The dashes print when there are no segment occurrences.

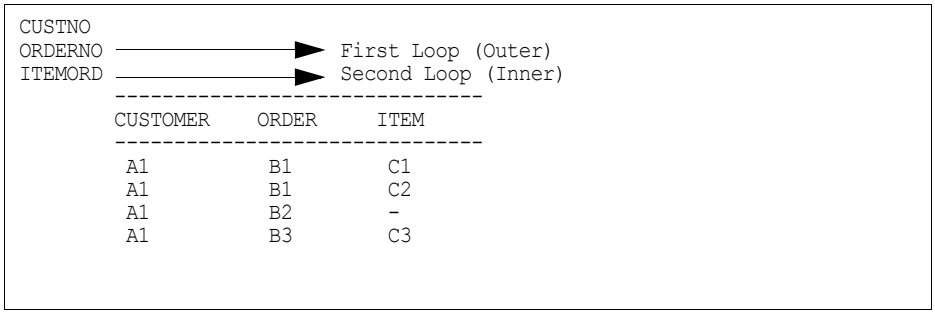

Figure 8-4 A Simple Nested Loop

Nested loops can be single path or multipath (see Nested Loops — Single Path and Nested Loops — Multi-path).

Note how the inner loop gets executed completely before the outer loop is executed. In this example, all occurrences of ITEM (C1 and C2) are output for the first occurrence of ORDER (B1) before proceeding to the second occurrence of ORDER (B2).

An important point to note is that the order in which the lower level segment fields are referenced makes a difference to the processing. If you reference the lower level segment types in an inverse order by referencing ITEMORD first and then ORDERNO (Figure 8-5), the outer loop is on the ITEMORD segment and the inner on ORDER.

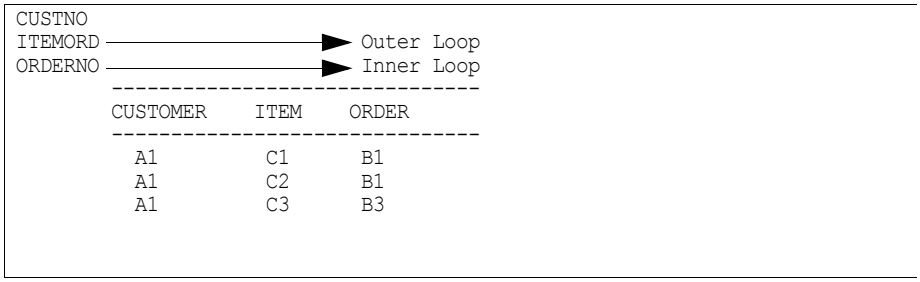

Figure 8-5 A Simple Nested Loop in Inverse Order

Note how the set of segments processed is different from that shown in Figure 8-4. The ORDER B2 is absent from Figure 8-4. This is because the ITEMORD segment is the outer loop and controls references to the inner loop. Orders without items are not processed.

Looping Rules

The following rules apply:

- The first executed reference to any field in a lower segment starts a loop on the occurrences of that segment type.
- The loop is executed for each segment occurrence of that segment type. Each occurrence is available to the procedure for further processing.
- The range of the loop extends to the end of the procedure.
- Nested loops (loops within loops) can occur by referencing different segment types.
- The innermost loop is executed first, the next outer, and so on to the outermost.
- A lower level segment type once referenced (thereby starting a loop), if referenced again will not start another loop in the same procedure. It is already in an automatic loop.
- Subroutine procedures will not start loops on segment types already part of a loop in the calling procedure.

Nested Loops — Single Path

Nested loops can be further subdivided into single path and multi path.

Single Path Nested Loop Definition

A single path nested loop is a nested loop generated by referencing multiple segment types along a single leg of the hierarchical structure. See Chapter 4, *Understanding Files* for a definition of a path or leg of a structure.

Figure 8-4 and Figure 8-5 are both single path nested loops.

Single Path Nested Loop Example

For example, given the single path shown in Figure 8-6, the procedure statements generate nested loops from the innermost segment type (last referenced) to the outermost segment type referenced.

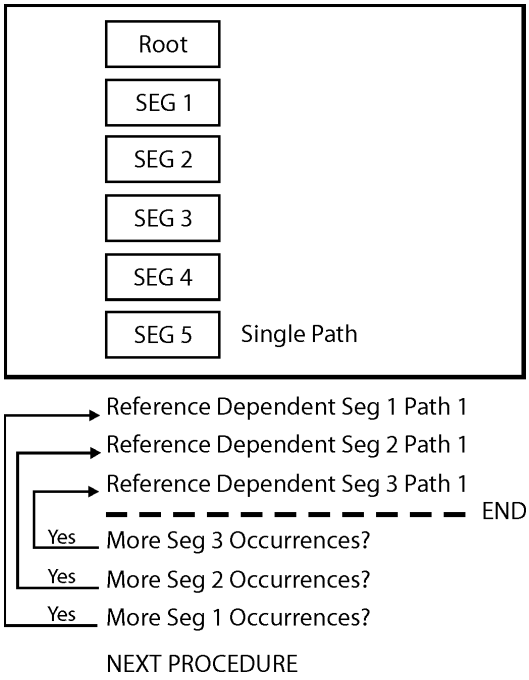

Figure 8-6 Single Path Nested Loop in Procedures

Note that loops are automatically controlled and processed to completion within a procedure. Branching locations within a procedure do not alter the looping process. For example, a CONTINUE branch causes processing to cease for the current occurrence of a segment in a loop and to resume (at the first reference to that segment type in the procedure) with the next occurrence of that segment type.

The automatic looping through lower level segments is one of the most powerful and convenient features of VISION:Inform. To use it efficiently, give careful consideration to where you first reference a lower level segment.

Suppose you have two procedures using the structure shown in Figure 8-2 and you make the following specifications.

PROC1

```
LET CUSTNO = . . .
LET ORDERNO = . . .
```

PROC2

```
LET ORDERNO = . . .
LET CUSTNO = . . .
```

If you write the calling procedure:

```
CALL PROC1
CALL PROC2
```

VISION:Inform loops through all occurrences of ORDERNO in PROC1 then returns to the calling procedure, then loops through all occurrences of ORDERNO in PROC2. For CUSTNO A1, this results in all three occurrences of ORDERNO being accessed twice.

However, if you start the loop on ORDERNO before you start the procedures, you loop only once for each occurrence. This is easily done by any reference to a field in the ORDER segment.

For example:

```
LET ORDERNO = ...
CALL PROC1
CALL PROC2
```

In this example, the loop on ORDERNO is performed once.

If you have a large database with numerous occurrences of the lower level segment, starting the loop in the calling procedure is much more efficient.

Nested Loops — Multi-path

Nested loops can be multi-pathed.

Multi-path Nested Loop Definition

A multi-path nested loop is a nested loop generated by referencing multiple segment types along two or more legs of one or more hierarchical structures.

Multi-path Nested Loop Example

For example, in Figure 8-7 the statements that reference ORDERNO and INSTNO start nested loops along two legs of the structure.

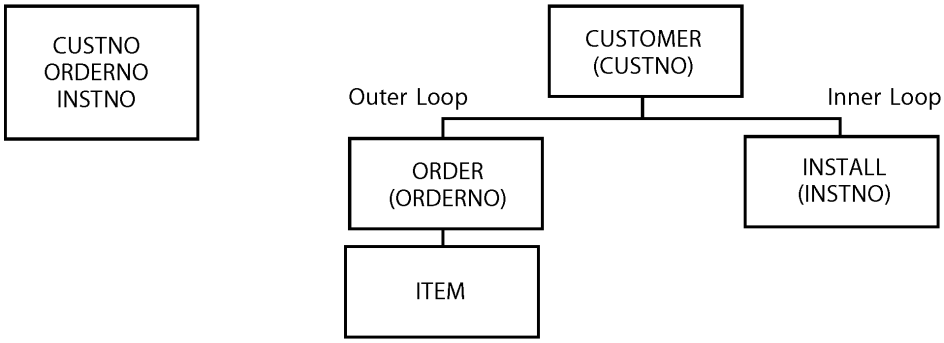

Figure 8-7 Multi-path Nested Loops

The fields that are referenced are shown in parentheses.

Note that under this definition, multi-path nested loops could also occur when references are made to fields in lower level segments in the old and new work areas or references are made to primary and synchronized file fields in lower level segments.

Flow of Control

Figure 8-8 shows the flow of control in the execution of multi-path nested loops. Note that the first statement that initiates a loop is the outermost loop

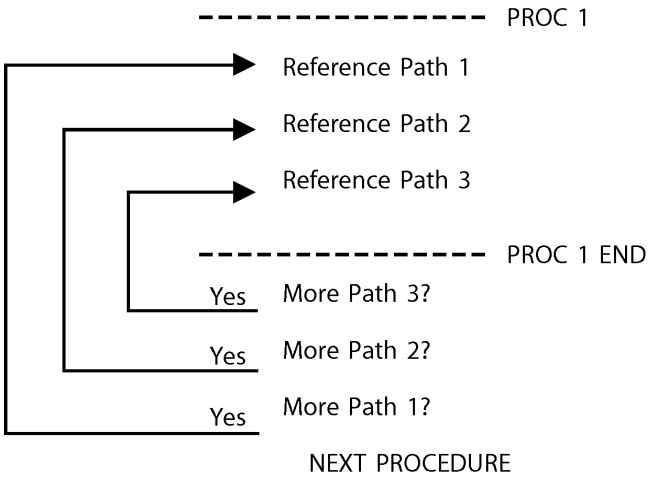

Figure 8-8 Multi-Path Nested Loops Flow of Control

Multi-path nested looping is not generally desirable unless you want all possible combinations of segment occurrences processed between the multiple paths.

Suppose you have this structure:

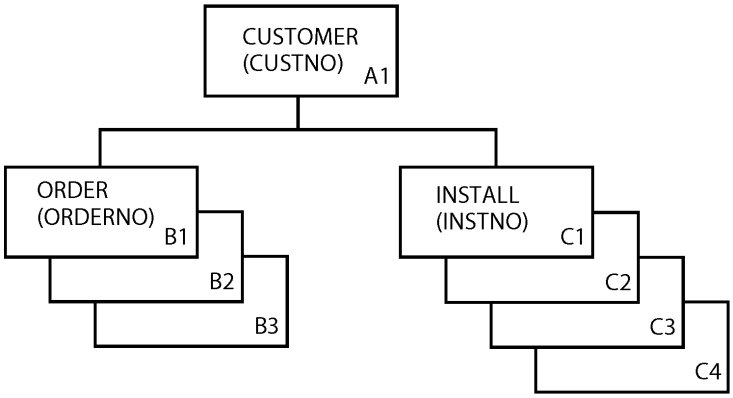

Figure 8-9 Nested Loop Structure Example

If you reference the following fields:

ORDERNO
INSTNO

VISION:Inform starts a loop on the ORDER segment then on the INSTALL segment. The set of data processed will look like:

ORDERNO	INSTNO
B1	C1
B1	C2
B1	C3
B1	C4
B2	C1
B2	C2
B2	C3
B2	C4
B3	C1
B3	C2
B3	C3
B3	C4

As you can see, this is not necessarily the results you expected. If the automatic looping process does not produce expected results, you can make modifications to produce what you want.

Standard Processing

A background processor processes queries and tasks against a database individually or in batches. The background processor optimizes processing by reading a database record and passing it to every query or task before reading the next record. How the database record is handled is determined by the type of processing specified.

The two types of processing available are standard processing and memory optimized processing. The default mode of processing is known as standard processing. In standard processing, the entire hierarchical record is built just prior to executing any queries or tasks. All the I/O for each database record is done before executing the batch of queries or tasks.

IMS Considerations

Note: The DBD structure must be within the limits of nine levels and 99 segment types.

With standard processing, you can create a file definition that describes all the segments in the DBD instead of just those in the PCB. With standard processing, if you reference a field in a segment that is not defined in the PCB, the reference is treated the same as a reference to a non-existent segment. The size of the IMS record determines the amount of memory required to process with standard processing. Standard processing reads the entire record into memory. It retrieves all occurrences of the segments defined in the PCB for each root; therefore, the larger the IMS record, the more memory required to process it.

Relational Considerations

VISION:Inform processes repeated lower level segments (such as multiple rows of tables) by issuing a SELECT in the form of OPENing a cursor (the OPEN of a cursor effectively does a SELECT).

Once the cursor is opened, VISION:Inform accesses rows of a table by issuing FETCH commands. When an OPEN (SELECT) is issued, it effectively commands DB2 to do I/O and to retrieve all the relevant rows of the table. A FETCH can be thought of as passing the data from DB2's memory buffers to the application. Thus, key to performance is the number of times a SELECT is issued and however many rows it finds to retrieve.

In standard processing, the hierarchical record is built just before passing the procedures against it. Since the I/O is performed outside of the procedure logic, all the I/O is done before executing against the record. Logical relationships have a great deal of flexibility in limiting the selection of records. Logical relationship statements are turned into WHERE clauses. Since several logical relationship statements can be included to qualify a particular segment, the use of these statements provides a more extensive WHERE clause. Thus, in this respect, it is possible to get better performance using logical relationship statements simply because there is better functionality to limit the number of rows retrieved.

GDBI Considerations

See Chapter 9, *GDBI Processing* for information on GDBI files.

Memory Optimized Processing

Memory optimized processing limits the amount of memory required to process a database. The savings occur as a result of accessing and storing in memory only one occurrence of each database segment type at a time. A segment is not accessed until it is referenced, unused segments are not accessed. There can be an increase in database I/O using memory optimized processing, as compared to standard processing if the segments are referenced in multiple procedures causing multiple reads of the same segment.

IMS Considerations

Unlike standard processing, with memory optimized processing and IMS, the file definition and the PCB must have a one-to-one correspondence for all segments. The DBD must contain a FIELD statement with a minimum of (SEQ,U) for every segment.

Relational Considerations

The difference in standard processing and memory optimized processing is the point in time a SELECT is issued. With memory optimized processing, data is obtained only when needed. Thus, if the logic of a query or task is such that the data is not required, it is not obtained (such as a cursor is not opened).

Furthermore, since the data retrieved for a subordinate segment is qualified by its parentage, far fewer rows of the table are accessed at any one time. While this appears to have an advantage, this advantage can be offset by the fact that all memory optimized processing may have to repeatedly access the same segment type. Consequently, there is a trade off between the number of rows accessed and the number of times this is done (such as the number of times that a cursor is opened).

GDBI Considerations

See Memory Optimized Processing with GDBI in Chapter 9, *GDBI Processing* for information about memory optimized processing with GDBI files.

From the discussions above you can derive the following:

- If you access every segment of a record (such as every row in every table), use standard processing.
- If your query or task uses a small subset of the segments in the database or the database record exceeds memory capacity, use memory optimized processing.
- If you batch queries or tasks, generally speaking, you should not use memory optimized processing.
- Use indexes to qualify the segments. The indexes should be the same fields that are defined as VISION:Inform key fields.

Processing Fields

The following sections describe processing for invalid fields, missing fields, and overflow fields.

Invalid Fields

During the course of arithmetic or logical operations or conversions, a field can become invalid for a variety of reasons:

- Illegal field data—non-numeric data where numeric data was expected; replacing a missing field (a field that does not exist in the record) into a result field.
- Division by zero.
- Field overflow—result fields not large enough to accommodate resulting data cause result fields to become invalid.

Flagging an Invalid Field

When VISION:Inform encounters an invalid field, it is flagged as invalid, but its original contents are not changed. This invalid property is propagated to other result fields which depend upon the invalid field, and they too become flagged as invalid. Again, field contents are not destroyed.

When there is an attempt to print an invalid field, the asterisk (*) character prints.

An asterisk also prints if a field is defined as a numeric field type in a file definition, but the data in the field does not match the definition. The asterisk, as an invalid field indicator, is the installation default.

Scope of the Invalid Property

The scope of the invalid property differs between ordinary database fields and system or temporary fields.

An Ordinary Field

A field which becomes invalid during a procedure remains invalid for the duration of that procedure only. Thus, any work area field which becomes invalid in procedure 1 and hence is not available to procedure 1, is available to subsequent procedures with no indication that it was ever invalid.

System Fields and Temporary Fields

The exception to this rule is that of temporary fields and system fields. Since temporary fields and system fields are designed to communicate between procedures, they carry their invalid property between procedures.

Re-validating Fields

All invalid fields can be re-validated by storing valid data into the field. Re-validation does not occur if the result field is partial-fielded during the store, even if the partial-field specification defines the entire field.

A field that is invalid before the first time it is referenced is not re-validated between procedures. For example, a record from a file contains alphabetic characters in a field defined as packed decimal. If the field is not re-validated, it remains invalid as long as that record is in memory.

A specific occurrence of a lower level segment is typically processed no more than once pre-procedure. It can however, be processed more than once through multi-path processing or multiple FIND operations.

- If a lower level field occurrence becomes invalid, it remains invalid as long as it is currently being processed.
- If it is re-processed through a subsequent loop or FIND operation in the same procedure, it will be valid at that time.

Subroutines and Invalid Fields

Subroutine procedures inherit fields that are in segment occurrences currently being processed by the calling procedure.

- If a subroutine procedure inherits an invalid field, the field can be re-validated and passed back to the calling procedure as valid. Similarly, an inherited field can become invalid in a subroutine procedure and is passed back to the calling procedure as an invalid field.
- If a non-inherited field becomes invalid in a subroutine procedure, it stays invalid only for the duration of that subroutine procedure.

Missing Fields (Nonexistent Fields)

Fields are said to be missing or nonexistent when either segment occurrences for the field do not exist in the record or its value is not available at a particular time during processing.

For example, if there are no occurrences of a defined segment type subordinate to the parent segment currently in memory, VISION:Inform treats the dependent segment as missing.

Missing fields display with a dash in the output (the dash is an installation option).

You can determine if a field is missing by testing if a field is equal to itself in a procedure statement. If missing, the test fails; if not missing, the test is true.

Replacing a missing field into result fields causes the result field to become invalid.

Invalid or Missing Fields in a Mapping Procedure

There is no such thing as a missing field in a mapping procedure. If a mapping procedure tries to reference a non-existent segment, VISION:Inform flags it as an error and terminates processing. Mapping procedures are only permitted to reference fields in the segment on which they are currently operating, or segments directly above along the same hierarchical leg.

For mapping procedures, as with standard procedures under non-memory optimized processing, fields are re-validated before each procedure begins execution. In memory optimized processing, a mapping procedure is executed within the application procedure, so it is neither practical nor wise to re-validate fields before each mapping procedure begins processing—data important to the application procedure would be lost if such a scheme were followed.

Field Overflow

VISION:Inform displays a + (installation option) for fields that are too large to fit in the output area.

Chapter

9

GDBI Processing

Generalized Data Base Interface (GDBI) is a feature of VISION:Inform that provides you with the capability to map data from virtually any Data Base Management System (DBMS) into a logical record structure that can be processed by VISION:Inform.

GDBI Concepts

This chapter covers the concepts and operation of the VISION:Inform GDBI.

- When using VISION:Inform with IBM databases (DB2, IMS, or VSAM), it is necessary to specify the logical record. VISION:Inform performs all databases accesses automatically.
- When using other databases, the access is through GDBI and it is necessary to provide procedures that will interface to the database manager. In essence, you are replacing the I/O portion of VISION:Inform with your own customized I/O calls.

VISION:Inform initiates a dialog with the GDBI linkage whenever it needs a logical record. Throughout the course of the dialog, the logical record is built one segment at a time using the related mapping procedures to perform the I/O to the DBMS file, returning segment data to the logical record buffer, until the logical record is completely built.

VISION:Inform then terminates the dialog and the logical record is available to the end user for processing.

The GDBI Linkage

Each file in the task or query and each file of a logical data view (LDV) that is to be processed through GDBI has a GDBI linkage defined for it.

The GDBI linkage is composed of three main components:

- GDBI file definition
- GDBI commands
- GDBI mapping procedures

Figure 9-1 shows the relationship among the three components.

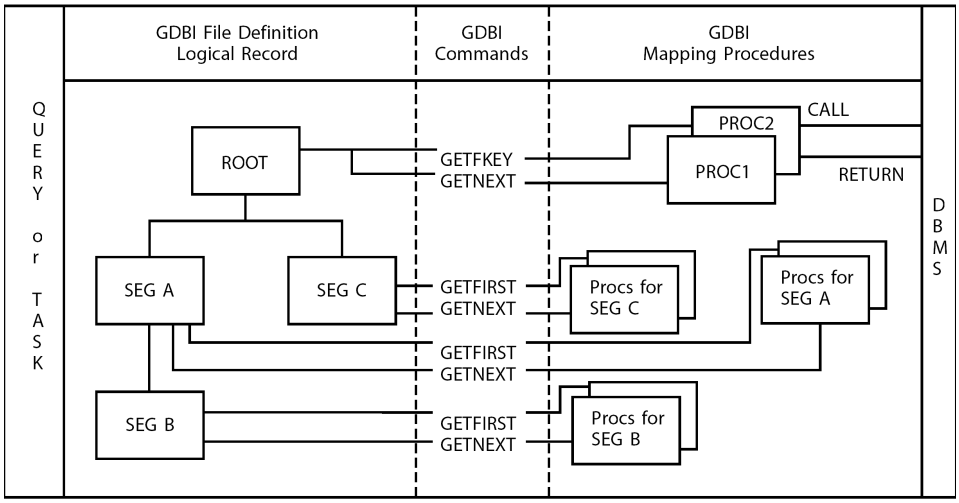

Figure 9-1 The Three Main Components of the GDBI Linkage

GDBI File Definition

The GDBI file definition is a structural entity that defines the logical hierarchy of the file to VISION:Inform.

GDBI Commands

For every segment in the file definition, there is a GDBI command that associates a mapping procedure with the segment. A command indicates the type of retrieval required, for example, GETFKEY (Get First by Key) or GETNEXT (Get Next). The segment name, command, and mapping procedure name tell VISION:Inform where to go to complete the I/O operation.

GDBI Mapping Procedure

The mapping procedure is a procedural entity that contains the calls to the DBMS or subroutines and moves the data for a segment into the logical record to pass to VISION:Inform.

GDBI Processing

When a user submits a query or task against a GDBI file, GDBI mediates between VISION:Inform and the DBMS. GDBI intercepts the VISION:Inform I/O operations and triggers the execution of the mapping procedures associated with the segments needed to build the logical records.

For example, when VISION:Inform begins to sequentially read a GDBI file, it issues an I/O operation to retrieve the first record. GDBI intercepts the I/O, looks at the GDBI file definition for that file to locate the mapping procedure associated with a GETFKEY command for the root segment, and calls the mapping procedure. It continues this process for each segment in the file definition until the record is completely built and then returns control to VISION:Inform.

Creating a GDBI Linkage

The following describes the steps for creating a GDBI linkage, and the dialog between VISION:Inform and the GDBI linkage:

1. A GDBI file definition that describes the logical structure of the database, as seen by the user application, is saved in the definition library and promoted to the foreground and background libraries. The GDBI commands for each segment are also contained in this file definition.
2. For each segment defined in the GDBI file definition, there exists one or more mapping procedures. The mapping procedures are also saved in the definition library and promoted to the background library.
3. When VISION:Inform needs a logical record, GDBI controls the process by building one segment at a time, using the mapping procedures to perform the I/O to the DBMS file and return the segment data to the buffer associated with the GDBI file.
4. When the logical record is completely built, it is available for processing by VISION:Inform.
5. Once the record is processed, the buffer is cleared and VISION:Inform begins building another logical record as in step 3, using GDBI to issue calls through the mapping procedures.

GDBI File Definitions

GDBI file definitions perform two functions:

- Define the logical structure of the record that the user's task or query will process.
- Bind the structure to the mapping procedures that eventually supply the data for the logical record.

This section discusses the conceptual relationship between the GDBI file definition and the mapping procedures.

GDBI file definitions are similar to standard VISION:Inform definitions in that they define a logical data structure to VISION:Inform and provide attributes of the data fields such as type and length. Like standard file definitions, they must be saved in the definition library and promoted to the foreground and background libraries.

The Uniqueness of GDBI File Definitions

The standard file definition describes a structured file with respect to the relative positions of the various segments in the data record. It directly corresponds to the physical record.

The GDBI file definition also describes a structured file, but the correspondence to the physical record is indirect. The GDBI file definition describes the logical record as the task or query sees it and does not necessarily have to correspond to the physical record.

The GDBI file definition can also define virtual segments. They provide work space that is available only to mapping procedures. See Virtual Segments for information about virtual segments.

The Purpose of Mapping Procedures

VISION:Inform “talks” to the DBMS through the mapping procedures which are bound to a segment by the GDBI Mapping Procedures panel of the Definition Processor.

- If a GDBI command does not need to be supported for a segment (for example, a keyed access is not done on the segment), then the procedure name is not required and can be left blank on the panel.
- However, if a GDBI command is issued and no procedure name has been provided, GDBI presumes that the segment was not found and continues building the record as if NFOUND was in the MSTATUS system field (discussed later).

The Relationship of the GDBI File Definition and the Mapping Procedure

Defining the GDBI file definition is a one-time operation in which the data files are described to VISION:Inform. The GDBI file definition defines the structure of the logical record. In addition, it associates the mapping procedures with a segment and a command. Many of these commands are issued multiple times, as needed, while VISION:Inform processes and builds each logical record.

GDBI Commands

There are four GDBI commands that indicate the type of retrieval required for a segment. They are:

Command	Function
GETFIRST	For root segments, this command is issued to retrieve the first occurrence of the root segment from the database. For non-root segments, this command is issued each time VISION:Inform moves processing from one segment type to another.
GETFKEY	This command is issued whenever the root segment of the database is retrieved with a key and is to be read sequentially from that point forward. It is used to position the database to the first record or to retrieve the first record within a range.
GETKEY	This command is only used when processing logical data views. It is used to retrieve the root segment by key value from the secondary database of the logical data view.
GETNEXT	This command is issued for retrieval of subsequent root segments after the first root segment has been retrieved with a GETFIRST or GETFKEY command. For non-root segments, GETNEXT is issued for retrieval of subsequent segments of a type subordinate to a particular occurrence of its parent type after a successful GETFIRST.

Initialization and Termination Mapping Procedures

In addition to the above commands, there are two other commands in a GDBI file definition that initiate two special types of mapping procedures—Initialization and Termination mapping procedures.

These mapping procedures are issued one time only for a file.

- The Initialization procedure is executed at the beginning of processing and is usually used to open the DBMS file being accessed.
- The Termination procedure is executed at the end of processing and usually performs the close functions on the DBMS file being accessed.

Commands	Functions
INIT	This command is issued once at the beginning of processing at the time when the GDBI file would be opened.
TERM	This command is issued once at the end of processing at the time when the GDBI file would be closed.

Virtual Segments

Virtual segments define a unique area of storage, called the virtual record, associated with an individual GDBI file. The area that they define is a flat, non-hierarchical area that is initialized only once, at the beginning of a VISION:Inform task or query. It carries information between records in a file, but not between files. As many as 99 different virtual segment types can be defined.

Using Virtual Segments to Collect Information

The virtual segments can be used to collect information about the file so that it is available to any mapping procedure for that file. For example, a mapping procedure may need to know the status of the last read or the number of records that have been read so far.

Using Virtual Segments as Intermediate Storage

The virtual segments can be used as an intermediate storage area for the current segment being read. When used in this fashion, the mapping procedure can move selected fields from the virtual segment into the logical record structure instead of moving the entire segment and save on the amount of storage required for the logical record.

Virtual Segment Names and Numbers

Since virtual segments are defined in relation to a specific GDBI file, the segment and field names that are used to define a virtual segment cannot duplicate any names used throughout the file definition. However, the same names can be used in different file definitions and VISION:Inform automatically associates each field with its appropriate file.

A virtual segment is defined using the GDBI File and Segment Definition panel. It is given a unique segment name and a segment number between 1 and 99. Since virtual and standard segments can have the same numbers, the segment number

alone cannot uniquely identify a segment. Instead, a new level of V is specified for a virtual segment. The level together with the segment number uniquely defines a segment.

Use of Virtual Segment Numbers

Segment numbers for virtual segments are merely a convenient notation; they are meaningless to VISION:Inform since they define a non-hierarchical area. However, since such numbers can have meaning to the user when writing a mapping procedure, VISION:Inform allows them. It is useful, for example, when you are reading each segment into the virtual segment prior to moving selected fields to the logical record. The virtual segment number can correspond to the standard segment number, making it easy to know which segment is contained in the virtual segment.

Virtual Segment Fields

Each virtual segment can have an unlimited number of fields. Fields within virtual segments are defined using the GDBI Field Definition panel, similar to defining fields within standard segments. Since the virtual segment record is a flat, non-hierarchical area, there are the following exceptions:

- Key fields are not permitted.
- Do not specify a count field nor a fixed occurs entry in a virtual segment.

Virtual segments cannot have an associated mapping procedure. Though they are defined in a GDBI file definition, there can be no relationship between a virtual segment and a mapping procedure.

A Sample GDBI File Definition

Figure 9-2 shows the Definition Processor panels for a GDBI file definition.

- The first panel is the GDBI File and Segment Definition panel.
- The second panel type is the GDBI Field Definition panel.

The remainder of this section discusses these specifications.

```
                          GDBI FILE DEFINITION              FILE   : VENDOR

   RECORD LENGTH          ===> 900
   MAPPING INITIALIZATION ===> OPENDB      FILE ID         ===> VENDTASK
   MAPPING TERMINATION    ===> CLOSEDB     EXPIRATION DATE ===> 12 / 31 / 99
                                           USER DATA       ===> VANESSA A.

Line      Segment    Hier     Seg    Seg       Procedure          Procedure
Cmd        Name      Level    Num    Ord    Command   Name     Command   Name
''''      VEND         1      001     _     GETFKEY  VENDGFKY GETFIRST _________
                                            GETNEXT  READVEND GETKEY   _________
''''      ORDER        2      002     _     GETFKEY  _________ GETFIRST READORDR
                                            GETNEXT  READORDR GETKEY   _________
''''      WORKAREA     V      003     _     GETFKEY  _________ GETFIRST _________
                                            GETNEXT  _________ GETKEY   _________
```

```
                          FIELD DEFINITIONS FOR
                      FILE: VENDOR     SEGMENT: VEND

                                                                      Seg  Seg No
Line  Primary   Alternate                        ... Field ...  Dec  Key  Count
Cmd   Fld Name  Field Name                       Len  Loc  Typ  Plc  Num  Field
''''  VENDSEG   _____________________________    87   1    C    __   _    ___
''''  VENDCTRL  _____________________________    7    1    C    __   1    ___
''''  VENDNAME  _____________________________    25   8    C    __   _    ___
''''  VENDADDR  _____________________________    30   33   C    __   _    ___
```

```
                          FIELD DEFINITIONS FOR
                      FILE: VENDOR     SEGMENT: ORDER

                                                                      Seg  Seg No
Line  Primary   Alternate                        ... Field ...  Dec  Key  Count
Cmd   Fld Name  Field Name                       Len  Loc  Typ  Plc  Num  Field
''''  ORDRSEG   _____________________________    43   1    C    __   _    ___
''''  ORDRCODE  _____________________________    2    1    C    __   1    ___
''''  ORDRVEND  _____________________________    7    3    C    __   _    ___
''''  ORDRONUM  _____________________________    5    10   C    __   _    ___
```

```
                          FIELD DEFINITIONS FOR
                      FILE: VENDOR     SEGMENT: WORKAREA

                                                                      Seg  Seg No
Line  Primary   Alternate                        ... Field ...  Dec  Key  Count
Cmd   Fld Name  Field Name                       Len  Loc  Typ  Plc  Num  Field
''''  VENDPJFR  _____________________________    87   9    C              ___
''''  VENDCTLX  _____________________________    7    9    C    __   _    ___
''''  ORDRBUFR  _____________________________    43   96   C    __   _    ___
''''  STATUS    _____________________________    4    1    C    __   _    ___
''''  REFER     _____________________________    4    5    C    __   _    ___
```

Figure 9-2 Definition Processor Panels for a GDBI File Definition

GDBI File and Segment Definition Panel

All the information about the file characteristics is entered on the GDBI File and Segment Definition panel.

File Name

The file name is VENDOR and the file is known to the DBMS as VENDTASK. VENDTASK is entered in the File Identification entry and will be available to the mapping procedures through the FILEID system field.

Mapping Procedure

The file is initialized with mapping procedure OPENDB and terminated with mapping procedure CLOSEDB. These procedures must be completed for this file to be processed.

Record Length

The record length specifies the size of the logical record and is provided so that VISION:Inform knows how many bytes to allocate for the buffer where a logical record is built. This estimate, 900 bytes in our example, must equal the largest expected size of the logical record.

Number of Occurrences per Segment

Since hierarchies can contain a variable number of lower level segment occurrences, estimate the maximum number of occurrences of each segment.

The number of occurrences per segment type must be multiplied by the size of the segment type. The result for each segment is then added up to obtain the estimated size of the largest logical record. See the section Memory Optimized Processing with GDBI for information on estimating the size of the logical record when memory optimization is used.

Root Segment

The root segment is named VEND and is assigned a segment number of 1. This segment occurs once per logical record. The VENDGFKY mapping procedure will handle the GETFKEY command for this segment, and the READVEND mapping procedure will handle the GETNEXT command.

Second Segment

The second segment is named ORDER. The mapping procedure READORDR will handle both the GETFIRST and the GETNEXT commands. READORDR builds either the first occurrence of this segment in each logical record or subsequent occurrences of this segment depending on the value in the COMMAND system field.

Third Segment

The third segment is named WORKAREA. It is a virtual segment and does not use a mapping procedures.

GDBI Field Definition Panel

All segment fields are defined on the GDBI Field Definition panel.

Root Segment Field Names and Key Field

The field named VENDSEG is defined for the whole length of the root segment, 87 bytes starting in position 1. This is a technique used to provide field access to the entire segment.

The key field for the VEND segment is named VENDCTRL. It is a 7 byte character field starting in the first byte.

The other VEND segment fields are VENDNAME and VENDADDR. VENDNAME references a 25-character vendor name. VENDADDR references a 30-character vendor address. All of the fields defined in the VEND segment occur once per logical record.

Second Segment

The ORDRSEG field is defined to provide field access to the entire ORDER segment which is 43 bytes long. Its key field, named ORDRCODE, is defined to begin in position 1 for a length of 2 bytes.

Virtual Segment

The last segment in Figure 9-2 is a virtual segment. It contains fields that would typically be used when calling a DBMS. It is called WORKAREA and is defined with a level of V. This segment will buffer data with the field named ORDRBUFR that is 43 bytes long. The segment maintains control and status information in the fields named VENDPJFR, VENDCTLX, STATUS, and REFER. The STATUS field contains return codes from the DBMS. The REFER field is used to provide key fields to the DBMS for a keyed read. Since this is a virtual segment, it is only available to the mapping procedures named in this GDBI file definition.

Defining Mapping Procedures

This section discusses issues that are germane to mapping procedures. However, you should read this section in conjunction with the chapter on procedures (Chapter 7, *Understanding Procedures*).

Once you have defined a GDBI file definition, you must provide mapping procedures to build its hierarchy of data. A mapping procedure is related to a segment and a GDBI command. When it is called, the mapping procedure is responsible for opening the DBMS file, closing the DBMS file, or retrieving data for the specified segment from the DBMS file and placing it into the logical record. The GDBI command indicates the function to be performed and the type of retrieval to be done.

A mapping procedure typically includes the following steps:

1. Sets up the parameters for the DBMS and calls it to perform the requested function.
2. Checks the status code returned by the DBMS.
3. Sets the MSTATUS system field to communicate to VISION:Inform the success or failure of the function, based on the status code from the DBMS.
4. Moves the data into the logical record if a retrieval was requested and was successful. Sets the MSTATUS field to NFOUND if a retrieval was requested, but the data was not found.
5. Returns control to VISION:Inform.

Using the Procedure Definition Panel

GDBI mapping procedures are created through the Procedure Definition panel, invoked by the Procedure option, in much the same way as any other procedure is created.

- Note that you must always enter N for the procedure type for those mapping procedures that were entered on the GDBI Mapping Procedures panel in the file definition.
- You enter S for other mapping procedures that are called from other mapping procedures.

Initializing a Segment

When a mapping procedure is called to build a segment occurrence for a file, VISION:Inform has constructed a "skeleton" segment in the logical record buffer. It contains blanks and/or zeros, depending on the field types.

A mapping procedure has access to only a certain subset of fields in the record it is building.

- All fields in the current segment are available and have been initialized to blanks and/or zeros.
- Any fields in the parent segments of this segment are also available and contain the data placed there by previous mapping procedures.

In other words, a mapping procedure has access to any segment defined in the same leg of the hierarchy as the current segment if that segment is at the same level or a higher level than the current segment.

A mapping procedure cannot access segments at levels lower than the one currently being built, nor can it access segments on parallel legs of the hierarchy. Fields from any other file are not accessible either.

Referencing Fields of Different Types — Qualifiers

A mapping procedure may have to reference fields of different types. When fields of different types are referenced, they are preceded with a single code to identify the source of the field (for example, system field, field in a file, temporary field). These codes are known as qualifiers. See Chapter 7, *Understanding Procedures* for information about qualifiers.

In a mapping procedure, a blank qualifier is used to reference any available field in the current file. Since mapping procedures are only used in a particular context with relation to a specific file, VISION:Inform automatically attaches a qualifier to a field of a GDBI file when it is referenced.

Referencing Temporary Fields

Any temporary field that is referenced in a mapping procedure is available to all the mapping procedures associated with all the GDBI files used in the task or query. They are not restricted to just one GDBI file and can communicate among GDBI files. However, they are not available to non-mapping procedures.

If a non-mapping procedure references a temporary field with the same name as a temporary field used in a mapping procedure, VISION:Inform maintains two distinct temporary fields—one for the mapping procedure and another for the non-mapping procedure. Each temporary field has its own distinct definition and value.

System Fields Used by GDBI Mapping Procedures

System fields are used in VISION:Inform to convey information between the task or query and VISION:Inform. In GDBI mapping procedures, there are additional system fields, available only to mapping procedures, that facilitate the communication among the mapping procedures, the task or query, and the VISION:Inform system. System fields are referenced in procedures with the qualifier F, for example, F.COMMAND, F.MSTATUS.

The additional system fields for GDBI are:

COMMAND An 8-character field which contains the name of the current GDBI command. The possible values are:

INIT	GETFIRST	GETFKEY
GETKEY	GETNEXT	TERM

FDNAME The file name of the GDBI file definition.

FILE The DD name of the file as it appears in the processing JCL. This contains the override DD name if one was specified.

FILEID The file identification that appears on the GDBI File and Segments Definition panel.

MODE Two characters that describe how the database is processed by VISION:Inform. The possible values are:

SR:	Standard processing
MR:	Memory optimized processing

MSTATUS A 6-character field used by a GDBI mapping procedure to instruct VISION:Inform to perform special actions when the mapping procedure returns control to VISION:Inform. The field is set to blanks when control is passed to a mapping procedure, which can then set it to one of the following values:

blank:	Successful retrieval
NFOUND:	Segment occurrence not found
STOP*nn*:	Tells VISION:Inform to stop processing the task or query and add *nn* to the condition code for the step.

PASSWORD An 8-character field that is set to the password provided on the Logical Data View Files panel or in the user profile binding the database to the task or query. Typically, this field is used by the initialize mapping procedure to open the database.

SEGNAME An 8-character field naming the segment which is currently being retrieved by the mapping procedure. LDV aliases never appear in this field.

System Fields with Separate Values

In addition, there are system fields that are available to both the logical data view procedures (LDV procedures) and the mapping procedures. VISION:Inform maintains separate values— one for the LDV procedures and another for the mapping procedures.

This enables the mapping procedure to reference the system field without disturbing the value set by the LDV procedures. They are commonly used in mapping procedures. These system fields are listed below. See also Chapter 7, *Understanding Procedures*.

CSTSTUS	MNUMBER	RNUMBER
LNUMBER	MSTART	RSTART
LSTART	RETURNCD	SSCOUNT

System Fields with Same Values

The following list of system fields is available to both LDV procedures and mapping procedures. However, VISION:Inform does not maintain separate values for these fields. They typically are not modified by procedures, but any changes that are made to these fields are apparent to both the LDV and mapping procedures. These system fields are listed below. See also Chapter 7, *Understanding Procedures*.

CHKP	JULIAN	TIME
CKPTID	LILIAN	TODAY
DATE	OWN	TODAYX
ISDATE	RESTART	All file count flags
JULANX	SQL	

Building a Logical Record with Mapping Procedures

GDBI intercepts the VISION:Inform I/O operation to read a record and initiates a dialog with the mapping procedures associated with the GDBI file to build a logical record. This section discusses the dialog that GDBI conducts with the mapping procedures in order to build a logical record.

How GDBI Constructs the Records

GDBI controls the building of a hierarchical record. Your mapping procedures do not need to know the order in which the segments are read into the hierarchy. GDBI constructs the record, segment type by segment type, one segment occurrence at a time, in a top-to-bottom, left-to-right fashion. Information from the GDBI file definition dictates the hierarchy and the related mapping procedures for each segment.

The dialog to build a logical record is initiated by GDBI when it intercepts the VISION:Inform I/O operation. It navigates through the hierarchy building segments one by one. For each segment, it constructs a “skeleton” segment that is initialized to blanks and/or zeros, depending upon the field types. It sets the appropriate values in the FDNAME, FILE, FILEID, SEGNAME, COMMAND, MODE, PASSWORD, and MSTATUS system fields.

Passing Control between GDBI and Mapping Procedures

These fields help point the mapping procedure in the right direction before it starts processing and informs GDBI of the results of the mapping procedure's actions. Then GDBI passes control to the appropriate mapping procedure. Which mapping procedure accepts control is determined by GDBI according to the type of I/O operation which is to be performed for the segment.

Once the mapping procedure has retrieved all the data, moved it into the logical record, and performed any required manipulations, it returns control back to GDBI with a return value set in MSTATUS.

These actions occur when either the mapping procedure reaches its final executable statement or when a CONTINUE statement is processed. In this way, mapping procedures are like subroutines to GDBI; they are called, executed, and then return control to the calling program at their termination.

GDBI continues the dialog by asking for repeated segment occurrences under a parent, until the mapping procedure returns a value of “NFOUND” in the MSTATUS system field. It then determines which segment in the hierarchy to build next and passes control to the appropriate mapping procedure.

When GDBI Terminates

GDBI terminates the dialog when all the occurrences for all the segment types have been built; that is, each mapping procedure for the lower level segments has returned a value of "NFOUND" in the MSTATUS system field. The record is completely built and GDBI returns control to VISION:Inform to process the record.

Figure 9-3 through Figure 9-10 illustrate how a record is built using standard processing and shows the order in which the mapping requests are invoked. The order of mapping requests for each segment are BUILD1, BUILD5, BUILD10, and BUILD15. The legend in Figure 9-3 lists some conventions which are used throughout the rest of this discussion.

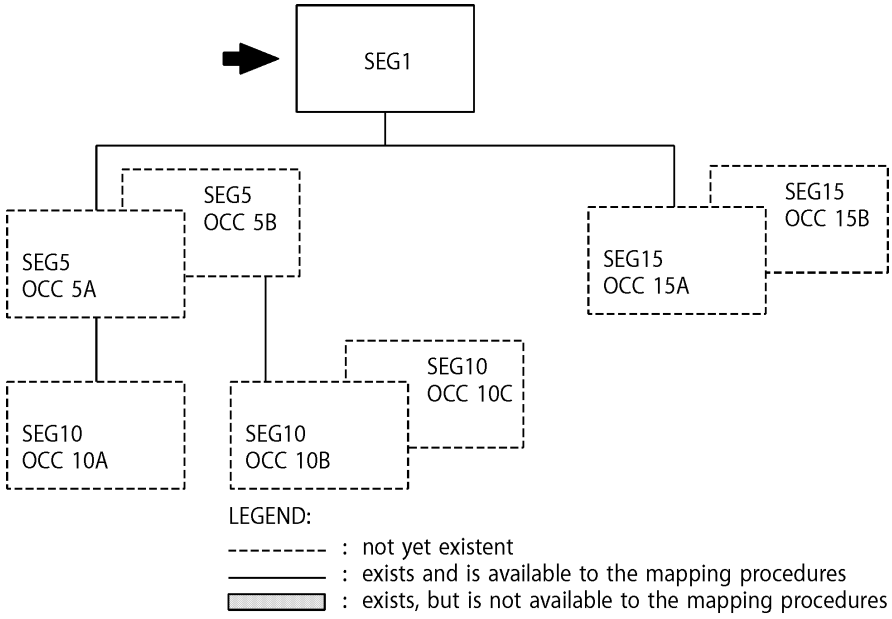

Figure 9-3 How a Record is Built (Page 1 of 8)

The first mapping procedure invoked is BUILD1, to build SEG1, with the COMMAND system field set to GETFKEY (Figure 9-3).

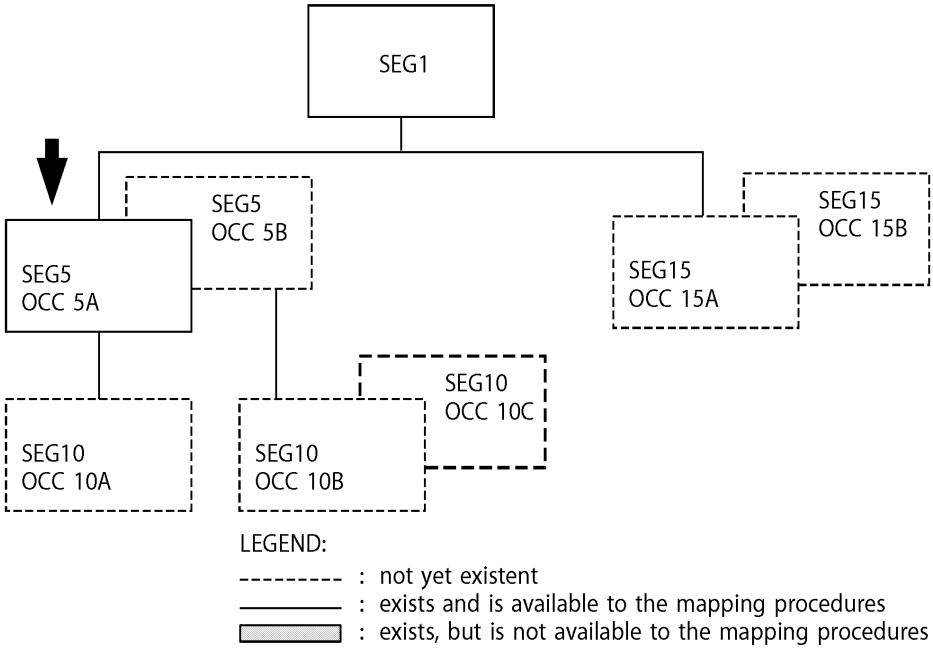

Figure 9-4 How a Record is Built (Page 2 of 8)

The next segment to be built is SEG5, so GDBI calls mapping procedure BUILD5 with COMMAND set to GETFIRST, shown in Figure 9-4.

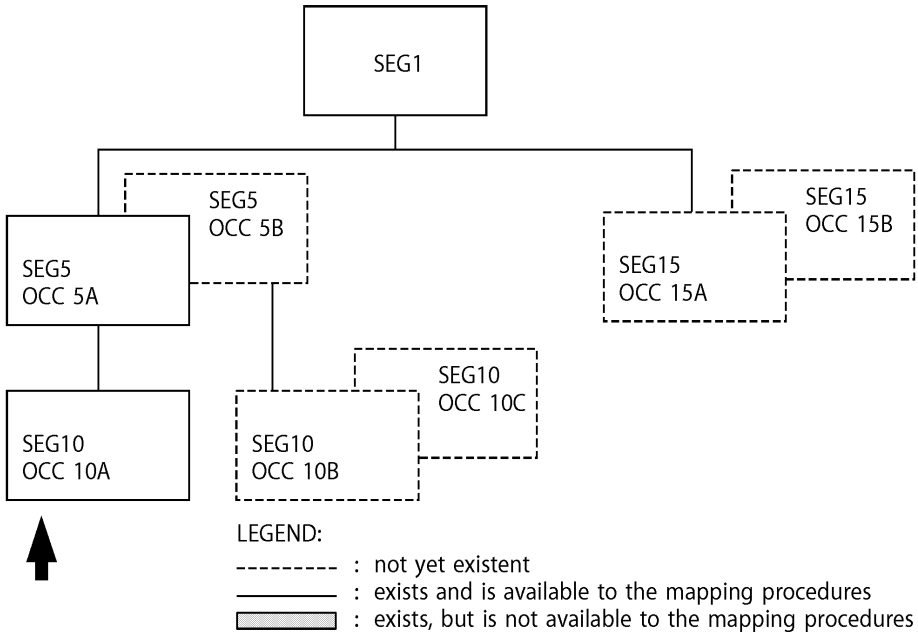

Figure 9-5 How a Record is Built (Page 3 of 8)

In Figure 9-5 the current segment has moved to SEG10 below occurrence A of SEG5. The COMMAND system field is GETFIRST. You will recall that record building progresses from top-to-bottom, left-to-right. GDBI tries to complete a single leg before moving on to repeated occurrences at higher levels.

Since GDBI knows from the file definition that there may be multiple occurrences of SEG10 and since GDBI also knows that SEG10 is at the base of its leg, it issues a second call to mapping procedure BUILD10. Since it is not requesting the first occurrence of the segment, the COMMAND system field is set to GETNEXT.

Mapping procedure BUILD10 returns a value of NFOUND in the MSTATUS system field which causes GDBI to retrace its path up the hierarchy until it reaches the first point at which it can take a branch to the right.

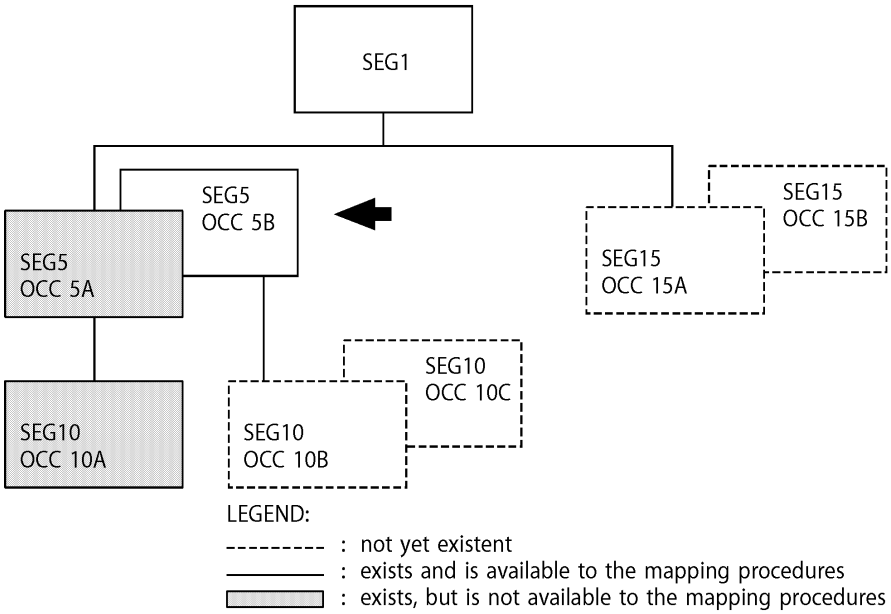

Figure 9-6 How a Record is Built (Page 4 of 8)

In this case, as Figure 9-6 shows, the opportunity arises at SEG5. Again, GDBI issues a call to mapping procedure BUILD5, with a value of GETNEXT in COMMAND, and is successful.

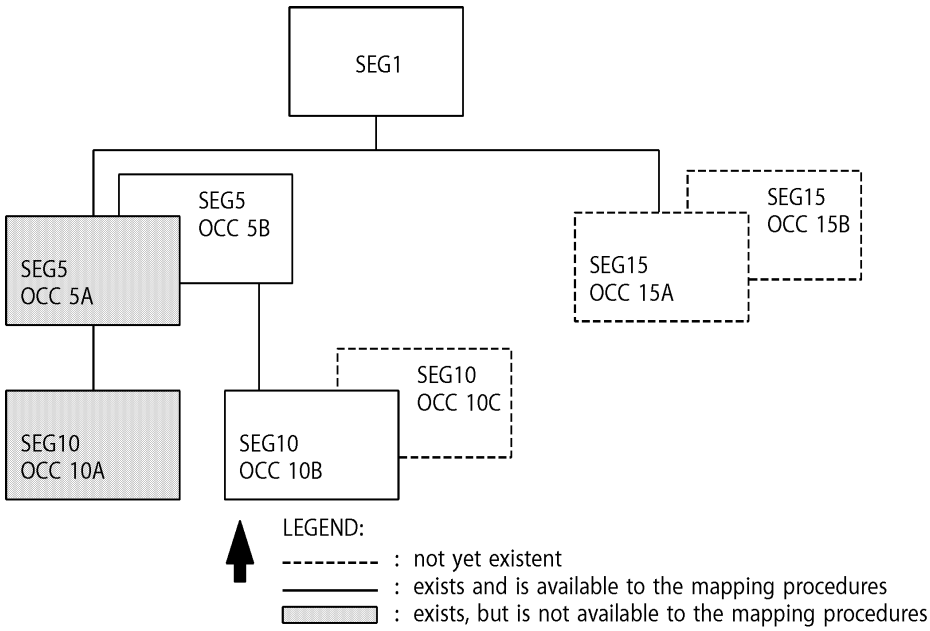

Figure 9-7 How a Record is Built (Page 5 of 8)

In Figure 9-7, GDBI has dropped down the second leg of the hierarchy to occurrence B of SEG10 under occurrence B of SEG5. Note that although occurrence A of SEG5 and occurrence B of SEG10 were successfully retrieved from the database and are now in memory, they are not available to mapping procedure BUILD10, or any other mapping procedure not involved with processing their leg of the hierarchy.

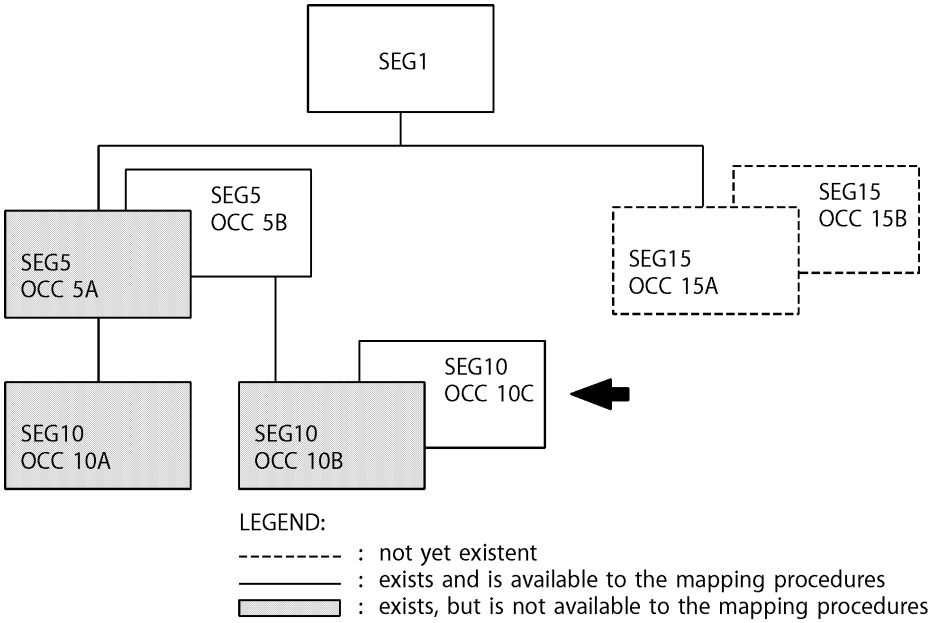

Figure 9-8 How a Record is Built (Page 6 of 8)

The remaining frames, Figure 9-8, Figure 9-9, and Figure 9-10, show the completion of the record building process.

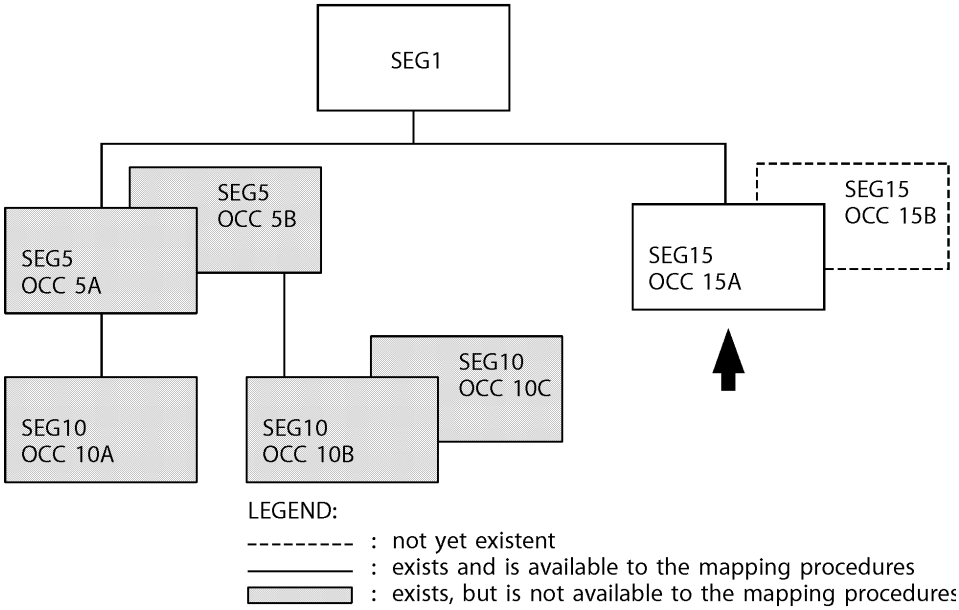

Figure 9-9 How a Record is Built (Page 7 of 8)

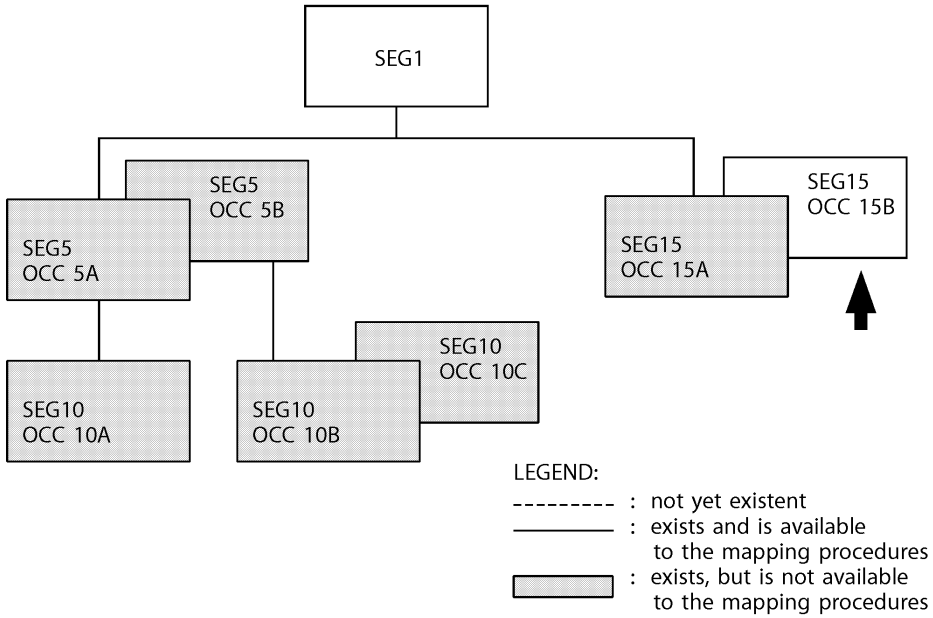

Figure 9-10 How a Record is Built (Page 8 of 8)

When the record is complete, GDBI is ready to return to VISION:Inform to process the record. A recap of the dialog follows.

Mapping Procedure	Segment	Command	MSTATUS	Occurrence Built
BUILD1	SEG1	GETFKEY	blank	the current root
BUILD5	SEG5	GETFIRST	blank	A
BUILD10	SEG10	GETFIRST	blank	A
BUILD10	SEG10	GETNEXT	NFOUND	none
BUILD5	SEG5	GETNEXT	blank	B
BUILD10	SEG10	GETFIRST	blank	B
BUILD10	SEG10	GETNEXT	blank	C
BUILD10	SEG10	GETNEXT	NFOUND	none
BUILD5	SEG5	GETNEXT	NFOUND	none
BUILD15	SEG15	GETFIRST	blank	A
BUILD15	SEG15	GETNEXT	blank	B
BUILD15	SEG15	GETNEXT	NFOUND	none

Responsibilities of the Initialization Mapping Procedure

Before the task or query is processed, the application needs to access the DBMS and open the database. These functions are performed by the Initialization mapping procedure that is started when the INIT command is issued at the beginning of processing at the point when files are opened.

- Specifying an Initialization mapping procedure name is optional.
- If specified, the first command issued under GDBI is INIT. GDBI selects the mapping procedure from the Initialization procedure entry on the GDBI File Definition panel.

There are some important things to know about the Initialization mapping procedure:

- It is invoked only once for the GDBI file, prior to issuing any other commands.
- It is named on the GDBI File Definition panel.
- All constant temporary fields requiring initialization should be initialized in this procedure.
- Most system fields are available, including COMMAND, FILE, FDNAME, FILEID, MODE, PASSWORD, CSTATUS, and RETURNCD.
- Only virtual segments defined in the GDBI file definition are available. Other fields in the file definition cannot be referenced since the record has not yet been built.
- Temporary fields defined in the mapping procedure set are available.
- The mapping procedure typically accesses the DBMS and opens the file or database for retrieval.

Sample Initialization Mapping Procedure

Figure 9-11 shows a sample Initialization mapping procedure named OPENDB. In this example, its responsibilities are to call the DBMS to open the database, check that VISION:Inform was able to call the DBMS, and analyze the return code from the DBMS.

- If either return code is inadequate, it sets the MSTATUS system field to tell VISION:Inform to stop the task or query with an appropriate condition code.
- Otherwise, it returns to VISION:Inform with the blanks in MSTATUS that were set prior to entry.

```
                PROCEDURE NAME: OPENDB          Procedure Type === N
                Reinit Temps? === Y             Maximum Items  === ____

;INITIALIZE A DATABASE
;
;ESTABLISH TEMPORARY FIELDS TO BE USED
;
RUNMODE: FIELD TYPE C LENGTH 2 INIT 'MT'
;
;OPEN THE DATABASE
;
CALL MODULE DBENTRY USING 'MULTOPEN' F.FILEID T.RUNMODE RESCODE
;
CHECK THAT IT OPENED CORRECTLY
;
IF F.CSTATUS EQ ' '   AND    RESCODE EQ 0 THEN
;  ALL O.K.  NO ACTION REQUIRED
ELSE
   LET F.MSTATUS = 'STOP20'    ;SOMETHING WRONG--SET ERROR CODE
END
```

Figure 9-11 Sample Initialization Mapping Procedure

Input Parameters

This hypothetical DBMS is called using a module named DBENTRY and requires input parameters typical of many systems. It needs a specification of:

- The particular type of open.
- The name of the database to be opened.
- A value indicating a run mode particular to the DBMS.
- A field which is used to contain the return code from the DBMS.

The order of the specifications is: 'MULTOPEN', F.FILEID, T.RUNMODE, and RESCODE.

Note that you can specify values either as literal values (such as 'MULTOPEN') or as field names containing the necessary value such as (T.RUNMODE.)

Location of the Database Name

The name of the database as defined to the DBMS was specified in the file definition and GDBI sets this value in the FILEID system field accordingly. It is communicated to the DBMS by a parameter on the call. This means that this mapping procedure does not need to know the name of the database, only that its name is in the FILEID system field.

Virtual Segment Field

RESCODE is a field defined in a virtual segment in the file definition. It is not preceded with a qualifier, signifying that it comes from the current file. This is a good example of using a field from a virtual segment to communicate among mapping procedures. RESCODE can be used by each mapping procedure to contain the result of the last access to the DBMS.

Checking CSTATUS

Checking the CSTATUS system field after each call to a module outside of VISION:Inform is a good practice. A call may be suppressed for a number of reasons (for example, an invalid parameter value) and subsequent checking of the called module's return code should be done only if a successful call was made. A value of blank in CSTATUS indicates a successful call.

Returning Control to VISION:Inform

When control is returned to VISION:Inform with a value of 'STOP20' in the MSTATUS system field, VISION:Inform adds 20 to the return code for the task or query and terminates.

In general, placing a value of 'STOPnn' into the MSTATUS system field tells VISION:Inform to set 'nn' to the return code and terminate. Choosing distinct numeric values for 'nn' helps pinpoint the exact error.

Responsibilities of the GETFKEY Mapping Procedure

Use the GETFKEY command whenever you need to position the database at a certain record and then read sequentially from that point forward.

VISION:Inform requests that the database be positioned at the beginning, or be positioned to a record with a particular key value. For an LDV, this command is issued only for the primary file.

Unlike the GETFIRST command, the GETFKEY command begins processing the database with a specific root segment. VISION:Inform places the key field value into the key field of the root segment in the record buffer prior to calling the associated mapping procedure.

The mapping procedure can then access it and pass it on to the DBMS by referencing the key field name. This command is issued only once for a GDBI file. All subsequent commands are GETNEXT commands since the file is read sequentially from the selected record forward.

The mapping procedure must check if there are zeros in the record buffer for the key field value. This means that VISION:Inform is requesting that the database be positioned at the beginning.

The following is a summary of what is required of the GETFKEY mapping procedure:

- The initial value of the root segment's key field must be obtained from the record buffer.
- If the key value is not zeros, pass the key value to the DBMS to establish position on that record.
- If the key value is zeros, inform the DBMS to position at the beginning of the database.
- If the DBMS call returns successfully, ensure that the retrieved segment is placed into the record buffer. Set the MSTATUS system field to blanks.
- If the DBMS cannot find the requested segment, set the MSTATUS system field to NFOUND.
- Return control to VISION:Inform.

Figure 9-12 shows a typical mapping procedure that would handle the GETFKEY command for our previously defined VENDOR database.

```
      PROCEDURE NAME: VENDGFKY         Procedure Type ===>> N
      Reinit Temps? ===>> N            Maximum Items  ===>> ____
;
;
;INITIALIZE THE RESULT CODE
;LET RESCODE = ' '
;
;
;POSITION AT BEGINNING OF DATABASE, IF KEY IS ZERO.
;IF VENDCTRL = 0  THEN
;
  CALL MODULE DBENTRY USING 'READPOS'  F.FILEID  'BEGINNING' ,
              RESCODE  T.DATAAREA
;
  IF RESCODE = '****' THEN         ;READ WAS O.K.
     LET VENDSEG = T.DATAAREA
  ELSE                             ;SOMETHING WENT WRONG
     LET  T.MESSAGE = 'READPOS ON VEND FAILED'
     LET F.MSTATUS = 'STOP30'
     CALL PROC TOTERR
  END
ELSE
;
;POSITION DATABASE BY KEY VALUE, IF KEY IS NOT ZERO.
;
  CALL MODULE DBENTRY USING 'READPOS'  F.FILEID  VENDCTRL ,
              RESCODE  T.DATAAREA
;
  IF RESCODE = '****' THEN         ;READ WAS O.K.
     LET VENDSEG = T.DATAAREA
  END
;
  IF RESCODE = 'MRNF' THEN         ;NO MORE RECORDS
     LET F.MSTATUS = `NFOUND'
  ELSE
     LET T.MESSAGE = `READPOS ON VEND FAILED'
     LET F.MSTATUS = `STOP31'
     CALL PROC TOTERR
  END
END
```

Figure 9-12 Sample GETFKEY Mapping Procedure

This mapping procedure, VENDGFKY, checks the key field VENDCTRL for zeros.

- If it is equal to zeros, the DBMS is called to position to the beginning of the database by passing the literal 'BEGINNING'.
- If the key field is not zeros, then the actual key value is passed to the DBMS by coding VENDCTRL as the third parameter.

In either case, the return code in RESCODE is inspected and if the retrieval was successful the data, which was returned in the temporary field DATAAREA, is moved to the VENDSEG field. Recall that VENDSEG defines the entire segment. Control is then returned to VISION:Inform.

Responsibilities of the GETKEY Mapping Procedure

The GETKEY command accesses a segment directly by key value. It is required for the root segment of a secondary file in an LDV, since this type of file is always accessed directly by key value. This command is never issued for the primary file in an LDV.

Unlike the GETFKEY command, the GETKEY command is issued for all root segments from the database. The GETNEXT command is never issued.

VISION:Inform places the key field value into the key field of the root segment in the record buffer prior to calling the associated mapping procedure. The mapping procedure can then access it and pass it on to the DBMS by referencing the key field name.

The following is a summary of what is required of the GETKEY mapping procedure:

- The initial value of the root segment's key field must be obtained from the record buffer.
- Pass the key value to the DBMS to retrieve the segment that matches that key value.
- If the DBMS call returns successfully, ensure that the retrieved segment is placed into the record buffer. Set the MSTATUS system field to blanks.
- If the DBMS cannot find the requested segment, set the MSTATUS system field to NFOUND.
- Return control to VISION:Inform.

Responsibilities of the GETFIRST and GETNEXT Mapping Procedures

The GETFIRST mapping procedure is called to retrieve the first occurrence of a particular segment type under its parent. The GETNEXT mapping procedure is called to retrieve each subsequent occurrence of a segment following a successful GETFIRST.

The distinction between the two commands is made in order to help set up parameters that may be required by the DBMS to establish positioning.

For VISION:Inform, the responsibilities of the mapping procedures for the GETFIRST and the GETNEXT mapping procedures are the same.

- Set up the required DBMS parameters and call it to retrieve a segment.
- If the DBMS call returns successfully, ensure that the retrieved segment is placed into the record buffer. Set the MSTATUS system field to blanks.
- If the DBMS return code indicates that no more segments exist, set the MSTATUS system field to NFOUND.
- Return control to VISION:Inform.

Figure 9-13 shows mapping procedure READORDR that handles both the GETFIRST and the GETNEXT command for the ORDER segment.

```
      PROCEDURE NAME: READORDR         Procedure Type ===>> N
      Reinit Temps? ===>> N            Maximum Items  ===>> ____
;PERFORMS A GETFIRST OR GETNEXT ON THE ORDER SEGMENT
;LET RESCODE = ' '           ;INITIALIZE THE RESULT CODE
'
IF F.COMMAND = 'GETFIRST'    ;GET FIRST OCCURRENCE UNDER PARENT
  LET T.DBOCCUR = 'FIRST'    ;TELL DBMS TO RETRIEVE FIRST OCCURRENCE
;
ELSE
  LET T.DBOCCUR = 'NEXT'     ;TELL DBMS TO RETRIEVE NEXT OCCURRENCE
;
END
;
;
CALL MODULE DBENTRY USING 'READS'  F.FILEID  T.DBOCCUR ,
            RESCODE  T.DATAAREA
;
IF RESCODE = '****' THEN         ;READ WAS O.K.
   LET ORDRSEG = T.DATAAREA
END
;
IF RESCODE = 'MRNF' THEN         ;NO MORE RECORDS
   LET F.MSTATUS = 'NFOUND'
ELSE         ;SOMETHING WENT WRONG
   LET T.MESSAGE = 'READPOS ON ORDER FAILED'
   LET F.MSTATUS = 'STOP40'
   CALL PROC TOTERR
END
```

Figure 9-13 Sample GETFIRST/GETNEXT Mapping Procedure

The COMMAND system field is checked in order to set up the proper parameter in temporary field DBOCCUR for either the GETFIRST or the GETNEXT command. Then the logic converges into a shared section of code that calls the DBMS and performs error checking. If a segment was retrieved, the MSTATUS system field remains blank. If a segment was not found, the MSTATUS system field is set to NFOUND.

Responsibilities of the Termination Mapping Procedure

When the task or query is done processing a GDBI database, it can then sign off of the DBMS and close the database. These functions are performed by the Termination mapping procedure that is started when the TERM command is issued at the end of processing at the point when files are closed. Specifying a Termination mapping procedure name is optional. If specified, the last command issued under GDBI is TERM. GDBI selects the mapping procedure from the Termination procedure entry on the GDBI File Definition panel.

There are some important things to know about the Termination mapping procedure:

- It is invoked only once for the GDBI file, at the end of processing the database, after the MSTATUS system field has been set to NFOUND for the root segment of the primary file. This means that there are no more logical records to process.
- It is named on the GDBI File Definition panel.
- Most system fields are available, including COMMAND, FILE, FDNAME, FILEID, MODE, PASSWORD, CSTATUS, and RETURNCD.
- Only virtual segments defined in the GDBI file definition are available. Other fields in the file definition cannot be referenced since no record is available at end-of-file.
- Temporary fields defined in the mapping procedure set are available.
- The mapping procedure typically signs off of the DBMS and closes the file or database.

Memory Optimized Processing with GDBI

You can specify memory optimized processing for a GDBI file in order to minimize the amount of storage needed to hold a logical record. Instead of holding all occurrences of a segment type in the record buffer, only one occurrence of a segment type is held in the record at a time.

- This means your task or query can process data from very large database records with substantial main storage savings.
- It also offers the possibility of minimizing the number of database accesses because only referenced segments are accessed, resulting in lower execution times.

You specify memory optimized processing for a GDBI file by entering a Y in the Optimize Memory? field on the Logical Data View Definition panel.

Standard Versus Memory Optimized GDBI Processing

Standard GDBI processing issues GDBI commands to build an entire logical record in the buffer. After the record is completely built in memory, the task or query processes the record.

In the case of memory optimized GDBI processing, only the command to build the root segment is issued prior to executing the task or query. The GDBI command to retrieve a lower level segment is not issued until the task or query references a field in that segment. The segment occurrence is read on demand into the record buffer by the appropriate mapping procedure, replacing the previous occurrence of the same segment type.

The result is that only one occurrence of any given segment type is in the record at a time. This means that the logical record size needs to be only as large as the sum of all the segment sizes. There is no need to multiply the size of a segment by the largest number of occurrences, since only one occurrence is in the record at a time.

GDBI Dialog for Memory Optimized Processing

Whereas the GDBI dialog for standard processing is completed prior to processing the task or query, the GDBI dialog for memory optimized processing occurs throughout the task or query, on demand, as segments are referenced. This is where the benefit of minimizing database accesses comes in.

- If a task or query does not reference any fields in a given segment, that segment is never retrieved.
- If there were 50 occurrences of the segment, a substantial savings is realized.

In standard processing, GDBI initiates a dialog to build an entire logical record and then returns control to VISION:Inform. However, in memory optimized processing, a GDBI dialog is initiated to build one segment occurrence and then returns to VISION:Inform.

In order to retrieve a lower level segment, it may be necessary for GDBI to retrieve the parent segments, if they have not been previously referenced by the task or query. So the dialog can actually consist of a number of GDBI commands in order to retrieve the desired segment. Your mapping procedure does not need to determine whether or not the parent segments are available. GDBI is responsible for this.

Memory Optimized Processing and the Database Characteristics

While memory optimized processing can save on database accesses, it can also increase them, depending on how the task or query processes the structure of the database.

- Memory optimized processing is suitable for a task or query that processes a restricted set of segments in a large database record and usually completely processes one segment type before moving to another segment type.
- It is not suitable when a task or query loops through the occurrences of a particular segment type a number of times or when multiple queries or tasks are batched together. Either of these scenarios cause all the segment occurrences to be retrieved again for each loop.

Writing Effective Mapping Procedures

Writing mapping procedures to effectively benefit from memory optimized processing does require a knowledge of the database as well as the application—in which segment fields are located, the hierarchical dependency of segments, and so forth.

The mapping procedure is coded in much the same way as it would be for standard processing. However, depending upon the DBMS, it may need to maintain database positioning information in order to hold position on a segment when the task or query references a segment in a different leg of the hierarchy.

The MODE system field is set to MR prior to calling a GDBI mapping procedure. This indicates that memory optimization is in effect for this file.

The example in Figure 9-14 illustrates a sample hierarchical record having five segment types at five levels.

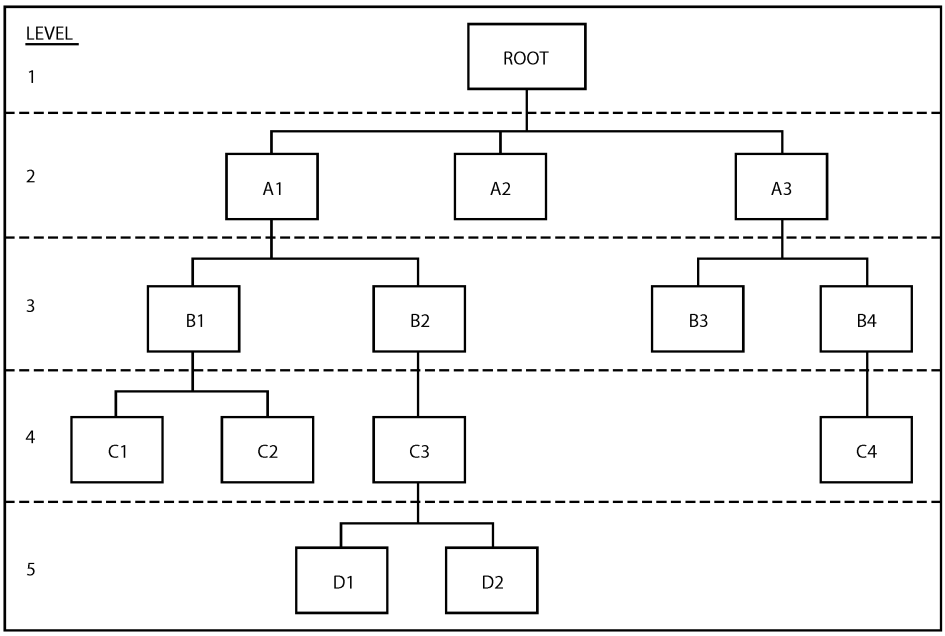

Figure 9-14 Sample Hierarchical Record

Assume that a task or query references only one field and it is in segment C. Its purpose is to retrieve all consecutive occurrences of segment type C and report them. The order in which the segments are retrieved with memory optimized processing is shown in Figure 9-15.

Notice that GDBI initiates five different dialogs in order to retrieve 4 segment occurrences.

- Segments A and B are retrieved in some dialogs, even though not explicitly referenced in the task or query, because they are parents of segment C.
- Segment D is never retrieved, because it is not referenced and is a child of segment C.

The segment occurrences are retrieved in the usual top-to-bottom left-to-right sequence.

The root is always available.

GDBI Dialog	Segment Level	GDBI Command	Value Returned in MSTATUS	Segments Available in Core	Return to VISION:Inform
1	2	GETFIRST	blank	A1	
	3	GETFIRST	blank	A1, B1	
	4	GETFIRST	blank	A1, B1, C1	RETURN
2	4	GETNEXT	blank	A1, B1, C2	RETURN
3	4	GETNEXT	NFOUND	A1, B1, C2	
	3	GETNEXT	blank	A1, B2	
	4	GETFIRST	blank	A1, B2, C3	RETURN
4	4	GETNEXT	NFOUND	A1, B2, C3	
	3	GETNEXT	NFOUND	A1, B2	
	2	GETNEXT	blank	A2	
	3	GETFIRST	NFOUND	A2	
	2	GETNEXT	blank	A3	
	3	GETFIRST	blank	A3, B3	
	4	GETFIRST	NFOUND	A3, B3	
	3	GETNEXT	blank	A3, B4	
	4	GETFIRST	blank	A3, B4, C4	RETURN
5	4	GETNEXT	NFOUND	A3, B4, C4	
	3	GETNEXT	NFOUND	A3, B4	
	2	GETNEXT	NFOUND	A3	RETURN

Figure 9-15 Order of Segment Retrieval with GDBI Memory Optimized Processing

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W