Warehouse User Guide

Taurus Software Inc.
420 Brewster Avenue
Redwood City, CA, 94063

(650) 482-2022
(650) 482-2010 FAX
support@taurus.com
www.taurus.com

First Edition.....October 1996
Second Edition.....April 1997
Third Edition.....October 1997
Fourth Edition February 2005

NOTICE

The information contained in this document is subject to change without notice.

Taurus Software, Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Taurus Software, Inc. shall not be liable for errors herein or for incidental or consequential damages in connection with the use of this material.

Table of Contents

Introduction 1
- **Product Overview** 1
- **How to Use The Warehouse Manuals** 3
- **Contacting Customer Support** 5
- **Product Version** 5
- **Manual Conventions** 6
- **Product Suggestions or Manual Corrections** 6

Product Concepts 7
- **Product Vision** 7
- **Product Uses** 9
  - **Data movement** 9
  - **Test environments** 10
  - **Archiving** 11
    - Archiving to offline media 12
    - Archiving to online historical databases 13
    - Rolling Archiving Strategy 14
- **Warehouse's Scripting Language** 15
  - Archiving Example 15
  - Retrieval Example 18
  - Establishing relationships with Warehouse 19
  - Data Movement 21
  - Test Environments 25

Platform Specific Information 27
- **Introduction** 27
- **HP3000** 29
  - **Running Warehouse on HP3000** 29
  - **Hints and Tips** 29
    - File Equations 29
    - Outputting to Tape on HP3000 30
    - Executing Warehouse in batch on the HP3000 30
    - Building Archive Files Before Archiving 30
    - Printing reports on the HP3000 31
    - Warehouse abnormal termination within a job stream 31
  - **Data Structures Supported on HP3000** 31
    - IMAGE 32
    - Oracle 33
    - Allbase 34
- **Unix** 35
  - **Running Warehouse on Unix** 35

Outputting to Tape on Unix....35
Printing reports on Unix....35
Data Structures Supported on Unix....35
Oracle....36
Windows....37
Running Warehouse on Windows....37
Printing reports on Windows....37
Data Structures Supported on Windows....37
ODBC....38
ORACLE....38

Warehouse Script Examples....39
Introduction....39
Database to Database Examples....41
Archiving to an online historical environment in an IMAGE environment....41
Moving multiple source databases to a similiar online historical environment.....44
Creating a summarized historical data file....50
Incremental loading of historical database....55
Script designed to be able to be restarted....58
Flat file used as selection for archival process....61
Maintaining logical data structures when moving from database to database....66
Checking records counts before moving data....70
Handling data conversion issues including logical data conversion changes....72
Using SQLNET to access a remote Oracle database....77
Freeze File Examples....79
Moving historical data to an offline freeze file....79
Moving historical data to an offline freeze file....83
Creating a historical archival log and moving data to a offline freeze file.....87
Retrieval historical data from a freeze file....90
Accessing multiple freeze files....93
Printing information from the freeze file....96
Using freeze files to create test environments....100
Mixed File Examples....103
Loading data from a flat file to an Oracle database....103
Using a flat file to drive a Warehouse process....110

Introduction

Product Overview

Up until or not so long ago, the applications being developed focused on collecting data and providing assistance with day-to-day functions, like taking an order, billing an order, and shipping an order. Today's environment includes requirements for keeping data for long periods of time, reformatting data for a number of different uses and creating subsets of data for test environments and data warehouses. What is needed is a way to selectively move data around paying special heed to data structure changes and handling those issues automatically.

Warehouse is a versatile tool which allows the selection of logical groupings of data, e.g. a purchase order and all its related information, and the movement of the data somewhere else, e.g. historical environment. With Warehouse you can select information from one or more data files, including IMAGE, Allbase, DB2, SQLServer, flat files or Oracle. These files can be physically or logically related. Based on selection criteria that you define, Warehouse moves the data to offline media, disc, or other data files.

Warehouse was initially designed as an archival tool to manage the needed but not always wanted volumes of historical data. Warehouse allowed you to selectively move historical data out of the way. Since its introduction in 1989, Warehouse's uses have expanded. We see our installed base using Warehouse in a variety of ways including: satisfying FDA requirements of lot related data, notifying customers of impending shipments of purchased goods through EDI, creation of test environments for integration testing of complicated and complex applications, and the creation of data warehouses.

If you are porting information from one platform to another you can use Warehouse's archive files to transport related data via one file and FTP it to the new location. Warehouse archive files have been designed to be platform independent. So, once the

data has been relocated, Warehouse will understand data stored in the archive file and move the data to your new file structure and deal with any data structure issues automatically.

Porting information is part of the data warehouse process. Warehouse can be an invaluable tool in extracting data from legacy systems, making the data consistent, transporting the data and its metadata to the integration process, populating the data warehouse and finally aging information out of the data warehouse. Warehouse's flexibility in dealing with logical groupings of data and file structural differences makes it a must have in this new emerging environment.

If you are archiving historical data, you can archive that information to tape or some other online historical environment. During the archival process, you can delete, update, and print information. Warehouse remembers the structure of your files, so if the structure of any file has changed since the archival process, Warehouse automatically handles the necessary data conversion.

If you are creating test environments, Warehouse allows you to create a test environment to handle a number of different test requirements. You can create them randomly or selectively. If your application requires linkages from data file to data file, Warehouse can select just the appropriate data and maintain those logical linkages. Select just that data needed to reproduce a problem and save that data for future regressive testing. Select data from the production version and move it into the new data structures under development, Warehouse will automatically handle any necessary data conversions. If necessary, Warehouse can even update or change the production data to shield sensitive data to be used in the test environment.

Experience the benefits of Warehouse's abilities to

archive and create test and reporting files. Enjoy extra disc space, reduced processing time due to fewer records to process, high quality test data, and easy data structure conversions through the use of Warehouse today.

How to Use The Warehouse Manuals

There are three manuals in the Warehouse documentation set. This may seem like a lot of documentation and you might be concerned the product is complicated to learn. Don't worry. Warehouse is built from twenty statements. It is easy to learn yet robust enough to handle the complicated problems that face your organization. To hasten the learning process for the variety of users using our product, we have designed manuals to try and match as many of the learning styles as we have encountered.

Each of the manuals is designed for the progressive learning that you will go through as you become familiar with the Warehouse product. Most of our users find the *Warehouse Tutorial* helpful as an introduction to the Warehouse product. Its hands-on approach to introducing the concepts and capabilities of Warehouse is very effective for the user that wants to get started right away.

The *Warehouse User Guide* is organized by the types of situations you might be trying to solve with Warehouse. The *User Guide* contains the big picture concepts including the design philosophy surrounding the Warehouse product. It also provides entire script examples solving real-life archiving, test database and data movement problems. These examples come directly from our install base. You will find techniques that may not be obvious in the plain syntax of each command.

If you are looking for the default values, exact syntax, or the handling of a particular data file or data type, you will find the answers in the *Warehouse Reference Manual*. In addition to the syntax of Warehouse's commands, you will find

answers to the syntax of the Warehouse's expressions, the rules regarding the processing of expressions, installation information and details regarding data file restrictions to each command.

Just starting? We suggest that read the *Warehouse Tutorial* first. Once you are comfortable with the concepts of the scripting language, the *User Guide* can provide you with some illustrative examples which may be helpful in developing your own procedures. For technical details, see the *Warehouse Reference Manual* for answers.

Warehouse User Guide

The *Warehouse User Guide* is structured to make learning Warehouse easy and painless. In Chapter 2, **Product Concepts**, the philosophy of the design of Warehouse are explored. In addition to this big picture of Warehouse, the uses of the product are explored and the components of the scripting language are introduced. Chapter 3, **Platform Specific Information,** describes specifics nuances regarding the uses of Warehouse on that particular platform. Chapter 4, **Warehouse Script Examples,** provides complete script examples for each of the major areas of use for Warehouse including: archiving, retrieval, test environment and the growing field of data movement.

The chapters of the *Warehouse User Guide* are as follows:

- Chapter 1 - Introduction
- Chapter 2 - Product Concepts
- Chapter 3 - Platform Specific Information
- Chapter 4 - Warehouse Scripts Examples

Since each chapter builds on information presented in earlier chapters, start reading at Chapter 2, **Product Overview**. Once you have read through the entire guide, specific information can be quickly accessed through the table of contents.

Contacting Customer Support

At Taurus, we have purposely bundled our updates and phone-in-consulting together as one product. We believe that our customer support is just as much a part of product as the documentation. We therefore encourage you to call them with any questions - whether it is about the correct syntax of the `PRINT` statement or the right archiving strategy for your company. You can contact them via email, telephone, or fax. To contact Customer Support by phone, dial (650) 482-2022 x2. To contact Customer Support by fax, dial (650) 482-2010. To contact Customer Support by email, address the message to support@taurus.com.

At minimum, the support representative, will need to know:

- Your problem
- Your platform and operating system
- Warehouse version
- How to contact you

Support Hours

Telephone help is available Monday through Friday from 8:30 a.m. to 5:00 p.m. Pacific Time (holidays excluded). For those with email, please send your information to support@taurus.com. If you do not have email, please fax copies of pertinent information to (650) 482-2010, attention Customer Support. Please remember to include your name, company name, and phone number on all messages.

Product Version

If you place a customer support call, it is important that you know the Warehouse version number that you are using. Since changes are made to Warehouse between each release, the Customer Support Representative may not be able to assist you properly without knowing the product version you are using.

The version number appears when Warehouse is run, and has three parts: major release (single character), update (number) and fix level

(numbers). `2.01.0001` is an example of a product version number with a major release of `2`, update level of `01`, and fix level of `0001`.

Knowing the correct version can save you time when calling Customer Support.

Manual Conventions

To make it easier to locate and understand topics in this manual we have used several conventions. The conventions are as follows:

- File names, job names, terminal response, and coding for job streams are shown in `courier type` font.
- References to keyboard functions are shown in upper case and bracketed with less than and greater than symbols (e.g. <RETURN>).

Product Suggestions or Manual Corrections

We have made every attempt to include the examples that will be the most helpful to extending your use of the Warehouse product. It is possible we have forgotten something that would help. It is also possible there are errors in this manual. If you find that either of these situations exist, please give us a call so we can correct it. We cannot continue to improve the documentation or the product without your input.

To submit a product enhancement or documentation change, either call customer support or use the form on our web site, http://www.taurus.com.

Product Vision

We have never seen product documentation which details the vision of the product from the developer's perspective. This section would not be in our documentation set if it was not for the requests of our users who wanted to get an overall understanding of how the product worked and how it was created.

Warehouse is written in C and is designed to be platform independent. The product is available for multiple platforms and for a collection of data file types.

When prospective clients ask us, "So, what does Warehouse do?" it is hard to answer because it can be used for so many different kinds of projects. Maybe the best way to answer that question is to say that at Taurus, we think of Warehouse as a data selection and movement tool. When we originally designed the product back in 1989, disc space was expensive and companies were looking for a way to manage their disc space better. Our investigation into how people were using their disc space showed us that 80% of the disc was being used by application data files. If companies were able to archive and delete obsolete data from their application data files, they could recover the disc space for other uses.

The industry has changed since 1989. Disc space is inexpensive. Now, companies say they are concerned with using their data to get the answers they need and they want to manage their data better. So, in 1993, in response to these new issues our customers and prospects were facing, we redesigned Warehouse from the ground up.

The projects that our prospects and clients were working on, whether they are creating test databases, removing historical information, or summarizing data for a decision support system, all have the same underlying problem -- how to select the data desired (regardless of the number of pieces it takes to make up the THING you want to move)

and move IT to the place desired without having to deal with differences in data structures between the sources and the targets.

Warehouse was designed to handle these issues. Warehouse determines what the targets look like, binds the related data which make up the objects, acts on those objects as a unit, and then moves the objects to your targets while dealing with any data conversion that may need to occur. Some examples of those conversions may be: changes in data mapping, changes in data types, changes in data lengths, new items, and missing data items. Warehouse handles all these issues automatically.

How does Warehouse accomplish this internally? Warehouse is written in modules. The kernel handles all generic functions like archive files, flat files and printing. The data file modules handle the read, writing, and locking required for each database subsystem. The data types for each database are supported in their native language. So when the data is moved from one integer type to another, Warehouse assures that the data will be stored in the target file as if it was originally created there.

The scripting language is designed to provide flexibility in describing relationships between data files, tables and datasets. There are a number of functions to help with the selection of data and to deal with the inevitable data type mismatches. There are constructs which allow you to override a subsystem's understanding of a data structure. There are programming constructs to help in dealing with complicated tasks like summarization and calculation. We feel that we have put together a set of constructs to handle the majority of data selection and movement projects you might encounter. However, we are always learning, so if you find an issue we haven't thought of, call customer support.

Product Uses

In earlier discussions, we mentioned that Warehouse is used by our clients for a wide variety of uses. This is true. However, the uses can be grouped into three major categories: data archiving and retrieval, creating test environments, and data movement projects. Let's examine each of these areas from both the user's and information service's perspectives.

Data movement

These days it doesn't seem to be enough that the data is available via the online transaction processing applications. More and more often there are demands for data extracted from the online transaction processing application be moved to another file or files for a variety of reasons. Some of our clients share their experiences.

A manufacturing and distribution company's most important clients requested notification of shipment of their orders. This notification would allow the client to notify the appropriate shipping and receiving departments and expedite receipt of these goods so they could be moved to the showroom floor as quickly as possible. To satisfy this requirement, our client needed to extract shipment information and notify their client via a standard EDI transaction that the shipment was on its way. This project involved selecting just the shipments for these special clients from the application database and consolidating and transforming the data to standard EDI transaction files. Warehouse provided them with the capabilities needed to satisfy this important client's request.

Another client said they needed a way to track expenses at the department level in order to determine what was being spent and why. Their data was organized by full account name. It was not possible to drill down through the data to get answers to what expenditures were made at each department level. What they said they needed was a way to extract data from multiple IMAGE sources

and populate a relational database. Warehouse extracted and transformed the data from the IMAGE database and copied the data directly into the Oracle tables needed. The accountant was then able to use his OLAP tool to find his answers from the new decision support system.

Another client had the need to create a training database which was based on data residing in an Allbase database on their production system. Unfortunately, the training database resided on another machine. What they wanted was a way to extract a small representative amount of data from the production application, change the name fields (to protect people's privacy), and put it into the Allbase database on the other machine. Warehouse provided them with the capabilities to do this project. Even though these two databases resided on different machines of the same platform, they could have just as easily have been machines of different types.

As you can see, our client's user's requirements for data span many different uses but the solution remains constant -- select the needed data, reformat it, and copy it to the new location.

Test environments

The next most common use of the product is in creating test environments. Creating test environments for applications which are complicated or have data which flows from one application to another is problematic for most information services departments. If you don't use a complete copy of all the data files, you run the risk of missing a problem in the flow of the data. However using an entire copy of the production environment makes validating the test results very hard because of the sheer volume of data you must deal with.

What Warehouse allows you to do is select small logically intact groups of data from the production environment and populate a test environment. You

get live "real" data. Because of the reduced size of the environment, you use less disc space, test runs are shorter, and validation of the data is easier. We had one client that experienced a 30% reduction in the errors that occurred when releasing a new version of the application software into production. In addition to the overall reduction of errors, the severity of production interruption (i.e. user functional downtime, e.g. production line shutdown because of the inability to enter work orders) was decreased.

There is no doubt that a good test environment is a necessary tool for any shop that does routine development. Now, with Warehouse, creating and refreshing a test environment is easy.

Archiving

Users require historical information for a variety of reasons. Some of the common reasons they cite are:

- Historical perspective is valuable when making decisions about where the company is going. Depending on the complexity of the issues and decisions you are trying to make, you may need the data organized in a different way.

- Resolving customer or vendor issues/disputes regarding payment or services provided. Depending on your business, research into a problem may require many years of history. The ideal access for these types of issues is online access -- maybe in a historical database.

- Regulatory or audit requirements. If the data is related in any way to your company's finances, you can be assured that the state and/or the federal government has the right to audit it for at least seven years.

In the best of all worlds, you could keep the data without any penalties. But there are penalties. Penalties and choices that the information service

organization is faced with daily. They include: degraded performance, long search times, long ad hoc report times, and ever expanding disc space requirements. When you find yourself faced with users having requirements for historical information and conditions like those described above, you could solve your problems with archiving.

So, where do you start? Well, you need to decide on an archiving strategy that is right for your company. The available strategies are: archive to online historical environment, archive to tape, or implement a rolling strategy where you first archive to a historical environment and then archive to tape. Let's review each of these strategies and their respective pros and cons.

Archiving to offline media

The most straight-forward way is to archive directly to tape. This methodology copies and deletes closed transactions to tape based on the selection criteria you provide. The archiving process can be done online or in batch. It can be run as often as you desire.

The result of the archiving process is an archive tape which contains the selected historical information. That information is deleted from the production environment. The deleted information provides additional capacity in the affected datasets. Processes which read through work orders serially run faster because there are fewer records to review. The historical information is available for reporting or retrieval if the need ever arises. Reporting can be done directly from tape.

If you ever need to retrieve the information from the offline storage, Warehouse will handle the data conversions that are necessary to bring that "old" data back into the current production environment. Warehouse can handle a number of data conversions automatically including: data type conversions, data mapping, data length changes, new data item initialization, and omission of non-

existent data items. This means you will never be stuck with archived data that you can't bring back and you will not be faced with a maintenance headache.

The disadvantage of this methodology is that all requests for historical information that has been archived to tape must come through the information services organization and requires someone to mount a tape to service that request. You must also be attentive to the length of time you are required to keep historical information as a tape's shelf life is only seven years. If the shelf life is a problem for you, you could archive to optical disc.

Archiving to online historical databases

The second option is to relocate historical information from the production environment into an online historical database. This database would be an exact copy of your production environment and would be accessible to your users via the application's usual inquiry and reporting transactions. Typically these environments are read only, so as to prevent users from inadvertently rewriting history.

The result of the archiving process is a production environment which contains only "active" information, maybe current year plus six months, and a historical environment which contains all other historical information. Performance for closing, extensive inquiries and reports in the production environment is dramatically improved. Access to the historical information is easy and uses the tools the end users are already familiar with. No additional training or special procedures need to be implemented. The archive procedures can be run as part of month-end close automatically and not impact the computer operations staff with tape mounts and special job scheduling.

The disadvantage of this methodology is that no disc space is saved. There is a one time initial

setup of the historical environment. Over time, you may want to archive to tape from your historical environment to make the volumes of data more manageable.

Rolling Archiving Strategy

The Rolling Archiving Strategy encompasses both of the archiving strategies described above. Historical information is first relocated from the production environment into the online historical environment. Once it stays there for some period of time, the historical information is relocated from the online historical environment to tape or some other offline storage.

The result of the archiving process is a production environment which contains only "active" information, maybe current year plus six months, a historical environment which contain historical information for some specified time frame and all other history is stored in the offline storage archives.

Performance gains are experienced for both the production environment and historical environment. Access to the historical information is easy and uses the tools the end users are already familiar with. No additional training or special procedures need to be implemented. Reports of the offline historical data can be produced without retrieving the information back on to the system, or, if needed the selected information can be relocated from the tape into the historical environment.

The disadvantage of this methodology is that no disc space is saved until you start to roll history offline. There is a one time initial setup of the historical environment. For information which has been relocated to offline storage, information service personnel will need to be involved in either retrieving or reporting of information.

Warehouse's Scripting Language

How does Warehouse accomplish this variety of tasks? Warehouse uses a scripting language which defines those records or groups of records and performs the appropriate actions on them. The scripting language is composed of three major parts: accessing files, selecting data, and data movement actions. To illustrate these components more clearly, let's review a series of Warehouse scripts.

Before we get into the details about the examples themselves, some general information about Warehouse and how the examples are presented might be helpful. Most scripts are developed in an editor and then executed in Warehouse via the `XEQ` statement. Scripts can be typed in uppercase or lowercase or both. Blank lines can be used. Comment lines are entered with `*` in position one and are ignored by Warehouse. If a statement needs to be continued over the limits of one line, use `&`. When Warehouse interprets the script, it numbers each line as it is processes it.

Archiving Example

Our first example is an archive example. We are going to archive all customers whose status is inactive. These inactive customers are copied to the archive file and then deleted from the Oracle database. The result of this archive process will be an archive file which contains only inactive customers. The Oracle database will contain no inactive customers.

```
1> open cust oracle scott/tiger
2> create inactc archive archive1
3>
4> read masters = cust.company_master for &
5>   status = "I"
6>   copy masters to inactc.company_master
7>   delete masters
8> endread
9> go
```

The first step in any script is to open the files you plan to access. In our script, in line 1 Warehouse opens the Oracle database using the user, `scott`,

and password, tiger. The database is given a tag, cust, which is used any time we need to refer to the contents of this database.

In line 2, Warehouse creates a new archive file, archive1, in the logon group and account. This new archive file is given the tagname of inactc. We will use this tag whenever we want to refer to the archive file. Line 3 is a blank line. Blank lines are helpful in making the scripts easier to read and understand.

Once the files have been opened, we need to tell Warehouse what we wish to select out of those files. Selection is done using the READ statement. Each READ statement is given a name or tag to refer to the data selected. Our READ statement took more than one line to enter, the statement is continued to the next line using &. In lines 4 and 5, the read statement tagged masters, reads records in the table company_master which have the value I in the column status from the database tagged cust. Notice that the database name tag is listed first followed by the table name, e.g. cust.company_master.

Please note that in order to ever execute lines 6 and 7, the record must have fulfilled the selection criteria in the READ statement. Line 6 copies the selected row to the archive file tagged INACTC. Notice that the actions like copy happen to tags. So in the COPY statement, we copy the masters tag to the archive file tagged, inactc and give it a name, company-master. You may name or tag the data whatever you please, but it makes sense to name it the same as your data source.

Line 7 deletes the selected records. Notice that the action, DELETE, always refers to the tag, masters.

Line 8 terminates this read block. All read blocks begin with a READ statement and end with an ENDREAD statement.

Warehouse will not process any data until it reaches a GO statement. Line 9, begins the Warehouse processing.

Retrieval Example

As a result of the execution of the previous script, we now have an archive file which contains inactive customers. To retrieve information from the archive file, you write a script.

```
1> open inactc archive archive1
2> open cust oracle scott/tiger
3>
4> read m = inactc.company_master for &
5>   cust_key = "1234"
6>   copy m to cust.company_master
7> endread
8> go
```

The first step in any script is to open the files. In line 1, Warehouse opens the archive file, `archive1` and tags it `inactc`. In line 2, Warehouse opens the Oracle database using the user, `scott`, and password, `tiger`. Warehouse tags it `cust`.

Line 3 is a blank line.

Lines 4 and 5 are the beginning of a read block. This read loop is tagged `M`. It reads the `company_master` table from the archive file tagged `inactc` for records that have `1234` as their value in the `cust_key` field. The `READ` statement didn't fit on one line so it was continued on the next line using the `&`.

Line 6 copies the selected record to the `company_master` table in the database tagged `cust`.

Line 7 terminates the read loop.

Establishing relationships with Warehouse

Unfortunately, most databases and the problems people are trying to solve are not this simple. The product needed a way to describe both logical and physical relationships between the data files. The way Warehouse establishes relationships is through the indented read loop. Let's expand our last example to archive the inactive customer and their ship-to-addresses.

```
1> open cust oracle scott/tiger
2> create inactc archive archive1
3>
4> read masters = cust.company_master for &
5>  status = "I"
6>    copy masters to inactc.company_master
7>    read ship-to = cust.ship_to_addr for &
8>      company-key = master.company_key
9>        copy ship-to to inactc.ship_to_addr
10>       delete ship-to
11>   endread
12>   delete masters
13> endread
14> go
```

Line 1 opens the Oracle database referenced by our Oracle home and SID and tags it `cust`. Line 2 creates the archive file, `archive1`, in the logon group and account.

Line 3 is a blank line.

Lines 4 and 5 build the read statement tagged, `masters`. This read loop selects records from the `company_master` table in the database tagged `cust` with the value of `I` in the `status` field.

Line 6 copies the selected records to the archive file tagged, `inactc`, and named or tagged `company_master`.

Our read loop tagged, `ship-to`, in lines 7 and 8 is our first example of an indented `READ` statement. Indented `READ` statements are how Warehouse establishes relationships. Indented `READ` statements are only executed if the above selection criteria have been met. So Warehouse only executes lines 7 and 8 if we have a customer that is inactive. Warehouse establishes the relationship

between masters and ship-to in the FOR clause of the READ statement. So, Warehouse selects the records from the ship_to_addr table in the database tagged cust matching the column company_key with the selected record from the masters READ loop with the value contained the company_key column in this table.

For each record selected, Warehouse in line 9 copies the selected ship-to record to the archive file tagged, inactc, and tags it ship_to_addr.

In line 10, Warehouse deletes the selected ship-to record from the database tagged cust. Notice that all data movement actions: COPY, DELETE, and UPDATE are done to tags.

Line 11, terminates the ship-to read loop.

Line 12, deletes the selected masters record.

Line 13, terminates the masters read loop.

Line 14, begins the Warehouse processing of the data.

Data Movement

Scripts for data movement projects are very different than either scripts for archiving/retrieval or test databases. This is because when you move data from one source to another, you typically want to change the organization of the data. In our next example the client wanted to move general ledger data from an IMAGE based transaction processing application to an Allbase based decision support system.

```
1> OPEN GLDB IMAGE GLDB PASS=; MODE=5
2> OPEN ADB Allbase ACCTDBE
3>
4> DEFINE ACBAL: RECORD
   1 -> GLACBALNUM : Allbase CHAR(24)
  25 -> GLACBALREGION : Allbase CHAR(2)
  27 -> GLACBALPRODUCT : Allbase CHAR(4)
  31 -> GLACBALCYAMT : Allbase DECIMAL(14,2)
  39 -> GLACBALLYAMT : Allbase DECIMAL(14,2)
  47 -> GLACBALUPD : Allbase CHAR(23)
  70 -> END
12>
13> DEFINE YEAR AS I1
14> SETVAR YEAR = 94
15>
16> READ ACCMAS = GLDB.ACCMAS FOR ACCTYP = 3
17>   SETVAR ACBAL.GLACBALNUM = ACCNUM
18>   SETVAR ACBAL.GLACBALREGION = &
19>      STR(ACCNUM,1,2)
20>   SETVAR ACBAL.GLACBALPRODUCT = &
21>      STR(ACCNUM,4,4)
22>   READ AMTFIL1 = GLDB.AMTFIL FOR &
23>      ACCNUM = ACCMAS.ACCNUM AND &
24>      AMTYR = YEAR
25>      SETVAR ACBAL.GLACBALCYAMT = AMTARRAY[1]
26>   ENDREAD
27>   READ AMTFIL2 = GLDB.AMTFIL FOR &
28>      FOR ACCNUM = ACCMAS.ACCNUM AND &
29>      AMTYR = YEAR -1
30>      SETVAR ACBAL.GLACBALLYAMT = AMTARRAY[1]
31>   ENDREAD
32>   SETVAR ACBAL.GLACBALUPD = &
33>      "1900-01-01 00:00:00.000"
34>   COPY ACBAL TO ADB.ARCHDB.GLACBAL
35> ENDREAD
36> GO
```

Line 1 opens the IMAGE database GLDB in the logon group and account using the database password ; and database open mode of 5. The database is tagged GLDB.

Line 2 opens the Allbase database environment ACCTDBE in the logon group and account. This

database is tagged ADB.

Line 3 is blank.

Line 4 begins the definition of a record type variable. This type of variable is very handy in creating a record which is composed of data items from more than one source. The variable works like a holding area for the data until it is time to write it out. When you begin a variable composed of many data items, Warehouse keeps track of the length of the records as you define each data item. In the script you can see the length indicator, e.g. 25->, displayed to the left of the data item. Warehouse generates this number. It is not contained in the script.

The next lines define each of the components of the ACBAL record variable. Notice that the record is defined using the native data types of Allbase as the record will eventually be written out to an Allbase table. The END signifies the end of the definition of this record variable.

Line 12 is a blank line.

Line 13 defines a variable named YEAR as an IMAGE one word long integer.

Line 14 initializes the variable YEAR to 94.

Line 15 is a blank line.

In line 16, the read loop tagged, ACCMAS, reads the ACCMAS dataset in the database tagged GLDB for records whose value in the ACCTYP is 3.

Line 17 sets the value of GLACBALNUM in the ACBAL record to the value contained in the data item ACCNUM in this record. So, we use the variable to hold the parts of the record that we need.

Lines 18 and 19 store the value contained in the first two positions of ACCNUM in GLACBALREGION in

the variable ACBAL.

Lines 20 and 21 store the value contained in the positions 4 through 7 inclusively in the data item ACCNUM in GLACBALPRODUCT in the variable ACBAL.

Lines 22, 23, and 23 define the read loop tagged, AMTFIL1, which reads the AMTFIL dataset in the database tagged GLDB for account numbers which match the current account number being read in the ACCMAS read loop and have the value of the variable YEAR in the data item AMTYR.

Line 25 sets GLACBALCYAMT to the value contained in the first instance in the array data item called AMTARRAY. The square bracket, e.g. [ ], defines which element of the array you want.

Line 26 terminates the read loop tagged AMTFIL1.

Lines 27, 28, and 29 are an example of the need for read loop tags. Read loop tags allow us to uniquely identify a set of records selected by a read statement. When you want to read the same dataset for two different conditions, if read loops didn't have tags, you wouldn't be able to identify which of these two conditions you are referring to when you copy or delete. Read tags allow this unique identification which is so critical when accessing the same dataset multiple times. Line 27 reads the data set AMTFIL (again) in the database tagged GLDB for the account number matching that selected in the read loop tagged ACCMAS and having a value in AMTYR matching the value in the variable YEAR minus 1. In other words, we are getting the same account but using last year's date.

Line 30 sets GLACBALLYAMT in the variable ACBAL to the value contained the first instance in AMTARRAY.

Line 31 terminates the read loop tagged AMTFIL2.

Line 32 and 33 sets GLACBALUPD in the variable ACBAL to the string listed.

Line 34 copies the completed record, ACBAL, to the Allbase table.

Test Environments

Scripts for test environments are not much different in substance from archiving scripts. As a rule, they differ only in that they: touch more of the database, only copy information, and must adhere strictly to the physical relationships imposed by the database structure. In our next example, a random sample of customer records is required. The user wants 100 customers and their associated ship-to and contact records.

```
1> open cust image custdb pass=READ mode=5
2> open test image custdb.data.test &
3>  pass=WRITE mode=3
4> define cust-cnt : i1
5> setvar cust-cnt = 0
6>
7> read c = cust.customer for cust-cnt < 100
8>    copy c to test.customer
9>    setvar cust-cnt = cust-cnt + 1
10>   read ship-to = cust.ship-to-addr for &
11>     company-key = c.company-key
12>       copy ship-to to test.ship-to-addr
13>   endread
14>   read contact = cust.encounter for &
15>     company-key = c.company-key
16>       copy contact to test.contact
17>   endread
18> endread
19> go
```

Line 1 opens the IMAGE database `custdb` in the logon group and account using the password `READ` and the open mode `5`.

Lines 2 and 3 open the IMAGE database `custdb` in the `data` group and `test` account using the open mode of `3` and the database password `WRITE`.

Line 4 defines a Warehouse variable. It is named `cust-cnt` and is an integer whose length is one word. For information on data types, see the *Warehouse Reference Manual* in the chapter entitled *Data Types*.

Line 5 initializes `cust-cnt` to the value of `0`.

Line 6 is a blank line.

In line 7, the read loop tagged `c`, reads the

customer dataset in the database tagged cust, checking the value of cust-cnt for each record to ensure that it is less than 100.

Line 8, copies records selected in the read loop tagged c to the customer dataset in the database tagged test.

Line 9, increments the cust-cnt variable.

Lines 10 and 11, are another example of an indented read statement. Remember that an indented read statement defines relationships between different datasets or data files. In this read loop tagged ship-to, Warehouse reads the associated records in the ship-to-addr dataset in the database tagged cust matching on customer-key.

Line 12 copies the selected record to the ship-to-addr dataset in the database tagged test.

Line 13 terminates the ship-to read loop.

Line 14 and 15 are another example of an indented read statement. In this read loop, contact, Warehouse reads the associated records in the encounter dataset in the database tagged cust matching on the customer-key read in the c read loop.

Line 16 copies the selected record to the contact in the database tagged test.

Line 17 terminates the contact read loop.

Line 18 terminates the c read loop.

Platform Specific Information

Introduction

Warehouse was designed to be non-platform specific. This has many advantages. The Warehouse that you run on UNIX has the same statements and features as the product you run on MPE/iX. However things like installation and program file names differ from one platform to another. This chapter deals with those differences.

HP3000

Warehouse runs on MPE/iX version 4.0 and forward. All software includes a single file, TAURUSWH.PUB.SYS, which once restored and run creates the TAURUS account and restores the Warehouse software and its support files into that account. For more information regarding the exact installation instructions, please see the *Warehouse Reference Manual*.

Running Warehouse on HP3000

To run Warehouse:

```
:RUN WH.WHII.TAURUS
```

If you have purchased additional database modules such as: Oracle or Informix, see the section on database specific information in this chapter.

Hints and Tips

C, the language that Warehouse is written in, was primarily a language used in the UNIX world. As a result of using C instead of a language originally developed for use on the HP3000 like COBOL or PASCAL, things don't always happen as you expect them to. This next section is meant to help you navigate through those differences as painlessly as possible.

File Equations

File equations are respected and can be used to point to another file. However, for the clarity of the script, it is better to fully "document" the location of the file in the OPEN statement. For example, if the file resides in the DATA group and you are currently logged onto to the PUB group, both of the examples below will work.

```
open datafile text datafile.data
```

or

```
:file datafile=datafile.data
:whii
open datafile text datafile
```

Outputting to Tape on HP3000

On other platforms, devices are not handled any differently than a file. When you write to tape drive, you are responsible for writing to that "file" and handling the consequences yourself. In order to make a product which could handle this philosophy, tape handling had to be modified slightly to run on platforms that have specific hardware drivers.

When writing to tape on the HP3000 use labeled tapes. The labeled tape will keep the volume set together, in case your archive spans more than one reel. Below is an example of the file equations needed to write to a labeled tape.

```
:file arctape;dev=dat;label=arc1995
:whii
1>open odb oracle fang/fum
2>create orcarc archive *arctape
```

Executing Warehouse in batch on the HP3000

Most of the work that you will do with Warehouse will happen in job streams. Running Warehouse in a job stream is the same as running it online. To reexecute Warehouse scripts, you may want to put the scripts in editor files and XEQ them. A sample job stream follows:

```
!JOB WHARC, MGR.PROD,DATA
!FILE ARCTAPE;DEV=DAT;&
  LABEL=ARC!!HPYEAR
!WH.WHII.TAURUS
XEQ ARCSCR
!EOJ
```

The Warehouse script resides in a file called ARCSCR in the DATA group in the PROD account. If you do not specify otherwise through the use of the REPORT file type, all Warehouse output appears in the $STDLIST.

Building Archive Files Before Archiving

Many customers do not want to archive to tape, but want to archive to disc. If you are one of these

customers, Taurus suggests building the archive file ahead of time. The reason for this is that if the system goes down before Warehouse has had the chance to write out the first block of data, the data will be lost. If the file exists prior to the archive process beginning, Warehouse will begin posting data immediately. If you need help determining how big to make your archive file, please call customer support and we will be glad to help with this. There is a discussion on estimating the size of the archive files in the next chapter, **Warehouse Scripts**.

Printing reports on the HP3000

When printing on the HP3000, you need to `OPEN` a `REPORT` file using the `OPEN` statement. The `OPEN` statement should use the `CCTL` option. You should also generate a file equation using `CCTL` to get proper page pagination. For example:

```
:file rptfil;dev=lp;cctl
:wh.whii.taurus
1> open rep report *rptfil cctl
```

For more technical information regarding this file type, see the *Warehouse Technical Manua*l. For examples of scripts using `REPORT` files, see the chapter entitled **Warehouse Script Examples**, in this manual.

Warehouse abnormal termination within a job stream

Warehouse always prints statistics at the end of a Warehouse execution regardless of whether the Warehouse execution is successful. However, Warehouse does change `JCW` to `FATAL`. You should be checking this JCW in your job streams before proceeding to the next step.

Data Structures Supported on HP3000

Warehouse supports the following data structures on the HP3000: IMAGE, Allbase, Oracle, fixed length flat files and fixed length text files. Some general information about the support of each of these structures:

- All file structure support is detailed in the **File Types** chapter in the *Warehouse Reference Manual*.
- For the most part all data file types are fully supported and supported at the intrinsic or API level.
- When adding, deleting, or updating data you must follow the rules required by the data file structure, e.g. IMAGE requires that before adding detail data that the associated manual master information must exist.
- Any logging or disaster recovery for the database or data file is your responsibility. If the subsystem has logging or IRL enabled, Warehouse's transactions will be logged.
- All critical data file errors cause Warehouse to terminate, e.g. data base corruption.
- All non-critical data file errors are reported by Warehouse.

IMAGE

Warehouse uses standard IMAGE intrinsics to access IMAGE. No privilege mode programming techniques are used to access IMAGE. By using this strategy in supporting IMAGE, Warehouse is fairly impervious to release changes of the operating system.

Warehouse honors the rules of adding, deleting or updating data in an IMAGE database. This means when deleting information from a master dataset all of the data in the associated detail datasets must be deleted first. This also means when adding data to a detail dataset that has manual masters associated with it, you must have the entries in the manual masters first. Details which have associated automatic masters add the data in the automatic master automatically whenever you add data into the detail dataset.

When you open an IMAGE database, Warehouse opens it using the mode and password you provide in the open statement. The password that you choose controls the data that you are able to access

in the IMAGE database. You should choose the mode compatible with the other processes that are accessing that database. For more information regarding mode compatibility, see the *TurboIMAGE Reference Manual*. If you do not specify a group or account, Warehouse looks in your logon group and account. Warehouse honors file equations. However for understandability, it is preferable to fully qualify the database name in the script.

Warehouse supports dataset, database, and item level locking. The locking mode that you choose should be compatible with the other processes that are accessing that database. If you are accessing multiple databases, use your locking conventions to ensure that you do not cause a deadlock situation to occur. How to turn specific locking options on and off is covered in the **Data File Types** chapter in the *Warehouse Reference Manual*.

Oracle

Warehouse uses standard API to access Oracle. No special modes or programming techniques are used to access Oracle. By using this strategy in supporting Oracle, Warehouse is fairly impervious to release changes of the operating system and Oracle releases.

To access an Oracle database using Warehouse, you must first set your Oracle SID and home system variables. This tells Warehouse which instance you are interested in and where Warehouse can find the database.

In the open statement, you provide Warehouse with a user and password. This user/password combination controls the security surrounding the different tables/column combinations. Warehouse provides you access only to those table/column combinations that your user/password security allows.

Warehouse honors any "rules" set out by the SQL

create statements used to create this database. If columns are NOT NULL or of a specific type, Warehouse ensures that these rules are followed. If any errors occur during the insert, Warehouse reports the error.

Allbase

Warehouse uses SQL to access Allbase. No special modes or programming techniques are used to access Allbase. By using this strategy in supporting Allbase, Warehouse is fairly impervious to release changes of the operating system and Allbase releases. As HP allows various releases of Allbase to run on the same operating system release, Warehouse is available for each of the releases. If you have questions as to which version you should be running, please call customer support at 650/482-2022 x2 or contact them via email: support@taurus.com.

HP only allows the access of one environment file per session. Warehouse allows you to access one in the session, plus any number of "remote" environment files. For more information about this either contact customer support or see the documentation on Warehouse server in the *Warehouse Reference Manual*.

Warehouse honors any "rules" set out by the SQL create statements used to create this database. If columns are NOT NULL or of a specific type, Warehouse ensures that these rules are followed. If any errors occur during the insert, Warehouse reports the errors.

Unix

Warehouse runs on a number of unix platforms and forward. Software is delivered on a tape in `tar` format. The installation process creates the TAURUS directory and restores the Warehouse software and its support files into that directory. For more specific installation instructions, please see the *Warehouse Reference Manual*.

Running Warehouse on Unix

To run Warehouse:

```
./taurus/whii/warehouse
```

Outputting to Tape on Unix

To write to tape, output the archive to the device desired.

Printing reports on Unix

When printing on the Unix, you need to `OPEN` a `REPORT` file using the `OPEN` statement. For example:

```
./taurus/whii/warehouse
1> open rep report rptfil
```

Once the file has been created, use `lp` to print it. For more technical information regarding this file type, see the *Warehouse Reference Manua*l. For examples of scripts using `REPORT` files, see the chapter entitled **Warehouse Script Examples**, in this manual.

Data Structures Supported on Unix

Warehouse supports the following data structures on the HP9000: Archive, Oracle, comma separated value (CSV) files, fixed length flat files and fixed length text files. Some general information about the support of each of these structures:

- All file structure support is detailed in the **File Types** chapter in the *Warehouse Reference Manual*.

- For the most part all data file types are fully supported and supported at the intrinsic or API level.
- All critical data file errors cause Warehouse to terminate, e.g. data base corruption.
- All non-critical data file errors are reported by Warehouse.

Oracle

Warehouse uses standard API to access Oracle. No special modes or programming techniques are used to access Oracle. By using this strategy in supporting Oracle, Warehouse is fairly impervious to release changes of the operating system and Oracle releases.

To access an Oracle database using Warehouse, you must first set your Oracle SID and home system variables. This tells Warehouse which instance you are interested in and where Warehouse can find the database.

In the open statement, you provide Warehouse with a user and password. This user/password combination controls the security surrounding the different tables/column combinations. Warehouse provides you access only to those table/column combinations that your user/password security allows.

Warehouse honors any "rules" set out by the SQL create statements used to create this database. If columns are NOT NULL or of a specific type, Warehouse ensures that these rules are followed. If any errors occur during the insert, Warehouse reports the error.

Windows Server

Software is delivered on a diskette and restored using the file SETUP.EXE. The installation process restores the software and its files into C:\Program Files\Taurus\Warehouse. For more specific installation instructions, please see the installation instructions located in the *Warehouse Reference Manual*.

Running Warehouse on Windows

To run Warehouse, double click on the Warehouse icon. Or for more control, run Warehouse in a command prompt.

Printing reports on Windows NT

When printing on the Windows, you need to OPEN a REPORT file using the OPEN statement. For example:

```
C:\>\Program Files\Taurus\WH MYSCRIPT.WH
1> open rep report rptfil.txt
```

Once the file has been created, you can use your standard printing methods to print it, e.g. right click on your .txt file that created. For more technical information regarding this file type, see the *Warehouse Reference Manua*l. For examples of scripts using REPORT files, see the chapter entitled **Warehouse Script Examples**, in this manual.

Data Structures Supported on Windows NT

Warehouse supports the following data structures on the Windows: Archive, SQL Server (ODBC), Oracle, ODBC level-2 compliant files, comma separated value (CSV) files, XML, fixed length flat files and fixed length text files. Some general information about the support of each of these structures:

- All file structure support is detailed in the **File Types** chapter in the *Warehouse Reference Manual*.
- For the most part all data file types are fully supported and supported at the intrinsic or API

level.
- All critical data file errors cause Warehouse to terminate, e.g. data base corruption.
- All non-critical data file errors are reported by Warehouse.

ODBC

No special modes or programming techniques are used to access SQL Server. By using this strategy in supporting SQL Server, Warehouse is fairly impervious to release changes of the operating system and SQL Server releases.

In the open statement, you provide Warehouse with user/password information. This password combination controls the security surrounding the different tables/column combinations. Warehouse provides you access only to those table/column combinations that your user/password security allows.

Warehouse honors any "rules" set out by the database definition used to create this database.

ORACLE

No special modes or programming techniques are used to access Oracle. By using this strategy in supporting Oracle, Warehouse is fairly impervious to release changes of the operating system and Oracle releases.

In the open statement, you provide Warehouse with Oracle user/password, Oracle home, and SID information. This password combination controls the security surrounding the different tables/column combinations. Warehouse provides you access only to those table/column combinations that your user/password security allows.

Warehouse honors any "rules" set out by the database definition used to create this database.

Warehouse Script Examples

Introduction

This chapter is here to help you understand the various facets of the Warehouse scripting language as it applies to the various data movement projects. This chapter is broken into sections with the examples grouped by movement types. The techniques illustrated in a movement type can be applied to any of the uses that you can imagine that fall within that movement type. Please don't let our lack of imagination interfere with your uses of Warehouse.

For example, the movement type of "database to database", could include a test database example, or moving historical data from the production database to the historical database or even include moving data from a legacy database into a data warehouse database.

Don't let your mind limit your understanding of the product by viewing the example for only one application.

Database to Database Examples

Script Type

Archiving to an online historical environment in an IMAGE environment.

Applicability

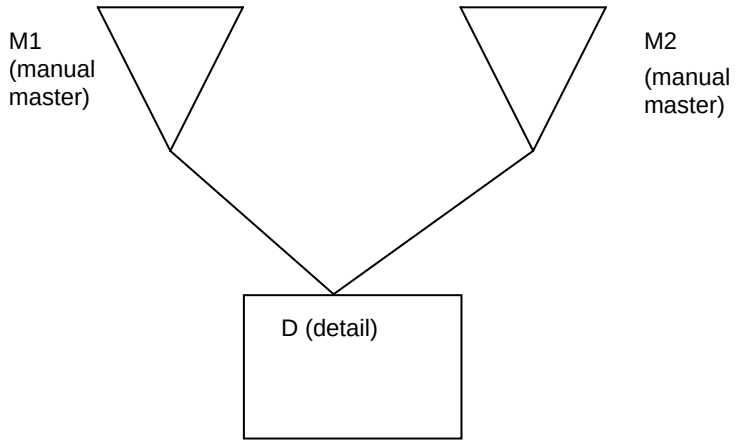
IMAGE to IMAGE example. IMAGE requires that manual master datasets must exist before you can add data to the attached detail datasets.

Technique Illustrated

Illustrates a technique used in populating a historical or test database. A detail dataset is tied to manual masters. This example illustrates how to populate the manual masters prior to adding to the detail dataset.

Background

The user wishes to extract information from a production database which has the structure illustrated below. The problem is TurboIMAGE does not allow information to be written to the detail dataset until both chain heads are in place, i.e., data must exist in both manual masters.



Script Explanation

The technique to handle this data structure is to read the detail dataset twice. The first read is used to establish the path to the second master dataset. Once the path is established, the second master's information can be copied to the test database. The second time Warehouse reads the detail dataset is to add the detail information to the output detail dataset. Please note that no deletions are done in

this script.

```
1>  OPEN PROD IMAGE PRODDB
2>  OPEN HIST IMAGE HISTDB &
3>   PASS=PASS MODE=3
4>  READ leftmstr = PROD. M1 &
5>   FOR item = 1
6>    COPY leftmstr TO HIST.M1
7>    READ DETAIL = PROD.D &
8>     FOR M1-KEY = leftmstr.M1-KEY
9>      READ rightmstr = PROD.M2 &
10>      FOR M2-KEY = DETAIL.M2-KEY
11>       COPY rightmstr TO HIST.M2
12>     ENDREAD
13>   ENDREAD
14>   READ REREAD = PROD.D &
15>    FOR M1-KEY = leftmstr.M1-KEY
16>     COPY REREAD TO HIST.D
17>   ENDREAD
18> ENDREAD
19> GO
```

Line 1 opens the IMAGE database `PRODDB` in the logon group and account using a default password of `;` and the default open mode of `5`. The database is tagged `PROD`.

Lines 2 and 3 open the IMAGE database `HISTDB` in the logon group and account using the password `PASS` and the open mode of `3`. The database is tagged `HIST`.

Lines 4 and 5 read the `M1` dataset in the database tagged `PRODDB` for data item `item` having the value of `1`. This read loop is tagged `leftmstr`.

Line 6 copies the selected record to the `M1` dataset in the database tagged `HIST`. Notice that you always use the tag name when using the `COPY`, `UPDATE`, and `DELETE` statements.

In lines 7 and 8, the read loop tagged `DETAIL`, reads the dataset `D` in the database tagged `PROD`, matching on the key previously found in the read loop tagged `leftmstr`.

In lines 9 and 10, the read loop tagged `rightmstr`, reads the dataset `M2` in the database tagged `PROD`,

matching on the key previously found in the read loop tagged DETAIL.

Line 11 copies the selected record to the dataset M2 in the database tagged HIST.

Line 12 terminates the rightmstr read loop.

Line 13 terminates the DETAIL read loop.

In lines 14 and 15, the read loop tagged REREAD, reads the dataset D in the database tagged PROD matching on the key from the read loop tagged leftmstr.

In line 16, Warehouse copies the selected record to the D dataset in the database tagged HIST.

Line 17 terminates the REREAD read loop.

Line 18 terminates the leftmstr read loop.

Comments

The technique can be used any number of times in the construction of a history or test database. Note: This technique is not necessary if the master datasets are automatic masters.

Database to Database Examples

Script Type

Moving multiple source databases to a similar online historical environment.

Technique Illustrated

Multiple source files to multiple target files. This technique could be applied regardless of data structure or data movement use. IMAGE to IMAGE.

Background

A school has two sets of databases:

- Student database which contains contact and demographic information.
- School year database which contains grade and attendance information.

Due to state funding, the school must be able to go back to a particular year's roster and provide attendance and student information for any dates within that year. In order to satisfy this requirement, the school has decided to create a historical database for each of the school years which reflects that year's activities.

So as we review the script, the technique being illustrated is how to link two data sources together in a script and copy them to two target databases. In this script look for the use of a report file and file equations.

Script Explanation

```
1>  open prodstu allbase stue
2>  open allyr image semstr &
3>   pass=WRITE mode=1
4>  open srpt report sturpt cctl
5>  open yr1993 image yr1993 &
6>   pass=; mode=3
7>
8>  header [srpt] $center, &
9>   "Students Transfer to Historical DB"
10> header [srpt]
11> header [srpt]
12>
13> read schyr = prodstu.prod.year for &
```

```
14>  year = "1993"
15>   copy schyr to prodstu.hist.year
16>   read student = prodstu.prod.stum &
17>    for school = schyr.school
18>     copy student to prodstu.hist.stum
19>     print [srpt], school, student_no,
20>     print [srpt], first_name,
21>     print [srpt] last_name
22>     read stud = prodstu.prod.stud &
23>      for student_no = &
24>      student.student_no
25>       copy stud to prodstu.hist.stud
26>       delete stud
27>     endread
28>     read grade = allyr.grades for &
29>      student-no = student.student_no
30>       copy grade to yr1993.grades
31>       delete grade
32>     endread
33>     read attend = allyr.attend &
34>      for student-no = &
35>      student.student_no
36>       copy attend to yr1993.attend
37>       delete attend
38>     endread
39>     delete student
40>   endread
41> endread
```

Line 1 opens the Allbase environment `stue` in the logon group and account. This database is tagged `prodstu`.

Lines 2 and 3 open the IMAGE database `semstr` using a password of `WRITE` and the database open mode of `1`. This database is tagged `allyr`.

Line 4 opens a report file named `sturpt` indicating that carriage control should be used. This report file is tagged `srpt`. If this script is running on an HP3000, a file equation needs to be present to point `sturpt` to the right print device and indicating that the file has carriage control, e.g. `:FILE STURPT;DEV=LP;CCTL`.

Lines 5 and 6 open the IMAGE database being used as the historical database named `yr1993` in the logon group and account using the password of `;` and the open mode of `3`. The database is tagged as `yr1993`.

Line 7 is a blank line.

Lines 8 and 9 define a header line to be printed at the top of every page on the report `srpt`. The text within the quotation marks is centered on the line. Note that you are able to have an unlimited number of report files. This means you could generate a report of all that was archived and a report of those records which didn't meet the selection criteria.

Line 10 generates a blank header line on the `srpt` report.

Line 11 generates another blank header line on the `srpt` report.

Line 12 is a blank line.

Lines 13 and 14 define the read loop tagged `schyr` which reads the table `prod.year` in the database tagged `prodstu` for records with the value `1993` in the field `year`.

Line 15 copies the selected record to the history table `hist.year` in the database tagged `prodstu`.

Lines 16 and 17 define the read loop tagged `student` which reads the table `prod.stum` in the database tagged `prodstu` for the corresponding records in this table matching on the `school` found in the previous read loop.

Line 18 copies the selected record to the history table `hist.stum` in the database tagged `prodstu`.

Line 19 generates a print line for the report `srpt` containing the fields `school`, and `student_no`. Notice the comma at the end of the line. This means

that the definition of this print line is going to be continued in the next PRINT statement.

Line 20 continues the print line for the report srpt started in line 19 and adds the field first_name to the print line.

Line 21 continues the print line for the report srpt started in line 19 and adds the field last_name.

Lines 22, 23, and 24 define a read loop tagged stud which reads the table prod.stud in the database tagged prodstu matching on the student_no found in the read loop tagged student.

Line 25 copies the selected record to the history table hist.stud in the database tagged prodstu.

Line 26 deletes the selected record.

Line 27 terminates the read loop tagged stud.

Lines 28 and 29 define a read loop tagged grade which reads the dataset grades in the database tagged allyr matching on the student_no found in the read loop tagged student.

Line 30 copies the selected record to the grades dataset in the database tagged yr1993.

Line 31 deletes the selected record.

Line 32 terminates the read loop tagged grade.

Lines 33, 34 and 35 define a read loop tagged attend which reads the dataset attend in the database tagged allyear matching on the student_no found in the read loop tagged student.

Line 36 copies the selected record to the dataset attend in the database tagged yr1993.

Line 37 deletes the selected record.

Line 38 terminates the read loop tagged `attend`.

Line 39 deletes the selected record in the read loop tagged `student`. This delete is done here because the record is needed to make the linkages to the related tables and sets. After all the associated information is deleted, we can delete the `student`.

Line 40 terminates the read loop tagged `student`.

Line 41 terminates the read loop tagged `schyr`.

Comments

This example showed the user moving data from a production environment to a historical environment. If you look past the specifics of this example, the techniques illustrated could be used in any database transfer project including: creating test databases, data conversion projects, or even a data warehouse project. So as we review the concepts illustrated, keep this in mind.

Two important concepts introduced in this script are Warehouse's capability to link multiple tables within the same database and the ability to make linkages to other databases or files. The linkages are made using indented read statements (a read statement which appears before its predecessor's endread) and through the `FOR` clause of the `READ` statement. The `FOR` clause describes the relationship. For example, `FOR STUDENT-NO = READTAG.DATAITEM`. In this example, `STUDENT-NO` is a data item in the table or dataset that is being read in this read loop. The information to the left of the equal sign refers to the data item that you wish to match on from a previous read loop. Warehouse doesn't care whether the linkage is a physical linkage as you would see in a master detail relationship in IMAGE or a logical relationship as a link from one file to another.

Warehouse allows the use of functions and expressions in the `FOR` clause of the `READ` statement. So if it were necessary to select pieces of two items and concatenate them together to make a linkage, you

could do that. For more information regarding Warehouse expressions, see the chapter entitled **Expressions**, in the *Warehouse Reference Manual*.

The next important concept introduced in this script is the use of report files. Report files are opened via the `OPEN` statement and then written to via the `HEADER` or `PRINT` statements. Each of these statements must refer to the report file to be written to. If no report file is specified, the `PRINT` or `HEADER` output is sent to the job's $STDLIST. The size of the report file is determined by the `SET` statement.

The last point is just a hint or tip. Notice that this script doesn't have a `GO` statement in it. With no `GO` in the script, Warehouse simply reviews the scripts and reports any syntax errors but does not process any data. This can be handy for debugging the syntax portion without having to worry about processing any data.

Database to Database Examples

Script Type

Creating a summarized historical data file.

Applicability

Summarization techniques. Could be used database to database or mixed files.

Technique Illustrated

Creating summarized data from detail transactions. Flat file to Oracle database.

Background

The company has detail transactions in a flat file and wants to create a historical database which contains just one record for each time we switch account or quarter. So assume the data source looks like:

Acct #	Qtr#	Amt	TxDate
1	1	5.00	1/1/91
1	1	1.00	2/1/91
1	1	4.00	3/1/91
1	2	10.00	4/1/91
2	1	5.00	2/1/91
2	3	5.00	9/11/91
2	3	5.00	9/17/91
2	4	10.00	12/1/91
		.	
		.	
		.	

In our summarized file, we would like to write:

Acct #	Qtr#	Amt	TxDate
1	1	10.00	
1	2	10.00	
2	1	5.00	
2	3	10.00	
2	4	10.00	

The key technique in this example is sorting a file and then writing records out which contain summarized information at those sort breaks. So, in the script below, look for the `order by`, which accomplishes the sort and copy of a record variable

instead of copying directly from the input source.

Script Explanation

```
1>  open transd fixed transd
2>  open sum oracle scott/tiger
3>
4>  format transf: record
    1 ->acct-no : image x(10)
    11->qtr-no  : image x(1)
    12->amt     : image i1
    14->tx-date : image z(8)
    22->comment : image x(40)
    62->end
11>
12> define sumrec : using sum.summary
13> ***Variables for checking for sort
14> *** break
15> define old-qtr : x(1)
16> define old-acct : x(10)
17>
18> define total-amt : i2
19>
20> * Initialize variables
21> setvar old-qtr = " "
22> setvar old-acct = " "
23> setvar total-amt = 0
24>
25> read tx = transd format transf &
26>  order by acct, tx-date, qtr-no
27>
28>   if qtr <> old-qtr or &
29>     acct-no <> old-acct THEN
30>     if acct-no <> " " then
31>       setvar sumrec.amt = total-amt
32>       setvar sumrec.comment = &
33>        '@sum record'
34>        setvar sumrec.acct-no &
35>         = old-acct
36>        setvar sumrec.txdate = &
37>         = tx-date
38>        setvar sumrec.qtr = old-qtr
39>        copy sumrec to sum.summary
40>        setvar total-amt = 0
41>     endif
```

```
42>     setvar old-acct = acct-no
43>     setvar old-qtr = qtr-no
44>   endif
45>   setvar total-amt = total-amt + amt
46> endread
47> if old-qtr <> 0 then
48>   setvar sumrec.amt = total-amt
49>   setvar sumrec.comment = &
50>    @Sum record"
51>   setvar sumrec.acct-no &
52>    = old-acct
53>   setvar sumrec.txdate = &
54>    = tx-date
55>   setvar sumrec.qtr = old-qtr
56>   copy sumrec to sum.summary
57> endif
58> go
```

Line 1 opens a fixed length flat file `transd`. The file is tagged `transd`. Two type of flat files can be used: `TEXT` and `FIXED`. `TEXT` files are to be used only when no numeric data exists in the file. If your file contains numeric data, use `FIXED`.

Line 2 opens the Oracle database using predefined variable values contained in Oracle home and SID variables. When the database is opened, you will be given access to the tables allowed access by the user and password supplied in your open statement.

Line 4 defines a format tagged `transf`. This record format defines how each of the various elements are read from the flat file. The definition of a format statement ends with an `end`. Notice that record definition shows the beginning of each field, e.g. the field `qtr-no` begins at position 11.

Line 12 defines a record variable, `sumrec`, which has the same layout as the summary table in the database tagged `sum`.

Lines 13 and 14 are comments. A comment is denoted by a `*` in column one.

Lines 15, 16, and 18 define local variables.

Lines 21, 22 and 23 initialize our variables.

Lines 25 and 26 read the transaction flat file, `transd`, using the format tagged `transf`. Notice that file is ordered by (sorted) by `acct`, `tx-date`, and `qtr-no`.

Lines 27 and 28 check to see if the account number or the quarter is not equal to our old account number or old quarter, if so lines from the `THEN` to the corresponding `ENDIF` are executed. If not, the logic is picked up after the `ENDIF`.

In line 30, Warehouse checks to see if the account number is not equal to spaces, and if so lines 31 through 41 are executed.

Lines 31 through 38 build the summary record from pieces of the transaction record.

Line 39 copies the summary record in the record variable tagged `sumrec` to the summary table in the database tagged `sum`.

Line 40 resets the `total-amt` variable back to zero.

Line 41 terminates the `IF` block.

Lines 42 and 43 set the old account number and old quarter to our current record.

Line 44 terminates the `IF` block.

Line 45 adds the current amount into the total amount field.

Line 46 terminates the `tx` read loop.

Line 47 checks if the `qtr-no` is not equal to zero. If it isn't, it executes the code within the block.

Lines 48 through 55 set the values in our summary records.

Line 56 copies the local variable, `sum-rec`, to our Oracle table, summary.

Line 57 terminates the `IF` block.

Line 58 begins the execution of the script. Please note that no processing of any data will happen unless there is a `GO` statement in your script. You can do syntax checking on your script by simply eliminating the `GO`.

Comments

Two important concepts were introduced in this script: using a local record variable to construct the data before writing it out to the database and sorting the file to induce breaking points.

Using a local record variable is very valuable in the following situations: where information has to be accumulated before writing the record out, where the information resides in multiple source tables, or where a single source table needs to populate multiple target tables.

Sorting a file with the `ORDER` by clause can also be used for printing a sorted report. The first item after the `ORDER` by is the highest sort item. The default sort order is ascending unless otherwise noted with `DESC`.

This example, as with all "archiving to historical environment" examples, illustrate techniques that could be used in migration, data warehousing, or even creating decision support environments.

Database to Database Examples

Script Type

Incremental loading of historical database.

Applicability

Could be used for test environments or data warehouses.

Technique Illustrated

Checking the target first to ensure that data doesn't already exist before copying the data over. IMAGE to IMAGE.

Background

The company has a number of code and description datasets. When they copy historical information over to the historical environment, they only want to select any new tax codes that may have been added since their last move. To accomplish this they read the entire source table and then check the target table to see if it already exists, if it doesn't exist they want to add a new table entry.

Script Explanation

The technique that is being illustrated is reading the target file and setting a switch if a record is found. In this example, Warehouse reads and writes to the "output" file, `hist`.

```
 1>  open prod image prod pass=W mode=1
 2>  open hist image hist pass=W mode=3
 3>
 4>  define copy : x(3)
 5>  read tax = prod.tax
 6>    setvar copy = "YES"
 7>    read outtax = hist.tax for &
 8>     tax-code = tax.tax-code
 9>      setvar copy = "NO"
10>    endread
11>    if copy = "YES" then
12>      copy tax to hist.tax
13>    endif
14>  endread
```

Line 1 opens our source database, `PROD`, which is

an IMAGE database. The database is tagged PROD.

Line 2 opens our target database, HIST, which is also an IMAGE database. The database is tagged HIST.

Line 4 defines a local variable, COPY, and defines it as an IMAGE X type 3 characters long. Notice, this wouldn't be allowed in IMAGE, but as a local variable we do not have to follow the rules outlined by IMAGE.

In the read loop tagged, tax, defined in line 5, Warehouse reads the tax dataset from the database tagged PROD.

In line 6, the copy switch is set to YES.

In lines 7 and 8, Warehouse reads the tax dataset in the target database, HIST, for the same tax code. If the read succeeds, Warehouse executes the code within the read loop, i.e. sets the switch in line 9. Line 10 terminates the read loop tagged outtax. If the read fails, the value of the COPY switch remains set to YES.

In line 10, we check if the switch, COPY, indicates if Warehouse should copy the record to the target database. If COPY is still set to YES, then in line 11, Warehouse copies the record to the dataset tax in the database tagged, HIST. Line 12 terminates the IF block.

Line 13 terminates the tax read loop.

Comments

The technique introduced in this example, has a number of different applications. Checking the target can be useful so that you don't duplicate data when bringing data back from an archive file or when copying data from the production environment to a test environment.

This example, as with all "archiving to historical environment" examples, illustrate techniques that could be used in migration, data warehousing, or even creating decision support environments.

Database to Database Examples

Script Type

Script designed to be able to be restarted.

Applicability

Universal, but assumes that it is running on HP3000.

Technique Illustrated

Building an archiving procedure that can be stopped and started without compromising the integrity of the logically related data. Allbase to Oracle.

Background

The customer has very large databases and very large archives (millions of records). Due to the type of business they are in, they need to have their databases available most of the day. The customer would like to be able to start and stop the archive process, but have it complete a full transaction before stopping.

In the example below, we see Warehouse continuing its execution of the script until a file appears. On the output block, A, Warehouse checks to see if a "stop" file, `stopfile`, has appeared. If the file has appeared, processing stops on the next read of the outer block. So to stop the archive, simply build the file.

Script Explanation

```
1>  OPEN PART Allbase PARTSDBE
2>  OPEN PARTARC ORACLE SCOTT/TIGER
3>
4>  DEFINE FLAG : I1
5>  SETVAR FLAG = 1
6>
7>  READ A = PART.MANUFDB.PARTS &
8>   FOR STATUS = "I" AND FLAG <> 0
9>    COPY A TO PARTARC.PARTS
10>   SETVAR FLAG = &
11>   SYSTEM("LISTF STOPFILE >$NULL")
12> ENDREAD
13> IF FLAG = 0
14>   PRINT "***Run aborted by",
```

```
15>   PRINT "STOPFILE***"
16> ENDIF
17> GO
```

The Allbase environment PARTSDBE is opened in line 1. This file is tagged PART.

Line 2 Warehouse opens the Oracle database using the user id, SCOTT, and the password, TIGER. The database is tagged PARTARC.

Lines 4 and 5 define a local variable, FLAG, and initialize it.

In lines 7 and 8, in the read loop tagged, A, the table MANUFDB.PARTS in database tagged PART is read for those records with the value of I in the column STATUS and for the local variable FLAG not having a value of zero.

Line 9 copies those records selected in the read loop tagged A to the PARTS table in the database tagged PARTSARC.

Lines 10 and 11 set the value of flag to the results of the LISTF. If the file is there, the value will be zero. If the file is not there, the LISTF fails and the returned value is not there.

Line 12 terminates the A loop.

Line 13 checks if the flag is zero.

If the flag was set to zero, lines 14 and 15 print a single print line which says ***Run aborted by STOPFILE***. Notice at the end of line 14, the comma indicates that print line should be continued beyond one line.

Line 16 terminates the IF block.

Line 17 begins the execution of the script.

Database to Database Examples

Comments

Warehouse has the capability to interact with the system it is being run on. The use of system variables to pass values for selection into the script is common. This script however, is using another technique to allow the user more control over the beginning and ending of the script.

The reasons for wanting to start and stop may be that there is limited time to archive. When archiving to a historical environment, you can begin and stop.

This technique could be used in a data conversion or migration projects.

Database to Database Examples

Script Type

Flat file used as selection for archival process.

Applicability

Using a flat file to drive a Warehouse process is a common technique. This technique could be used in creating test environments, creating data warehouse or simple data movement as in EDI. Flat file and IMAGE to IMAGE.

Technique Illustrated

Illustrates using a flat file to drive an archive process. The flat file contains key values of those records which have been qualified for archiving. This flat file could have been created by another process or program such as SUPRTOOL or SQL. This technique is also useful for creating a test database containing a predetermined list of records.

Background

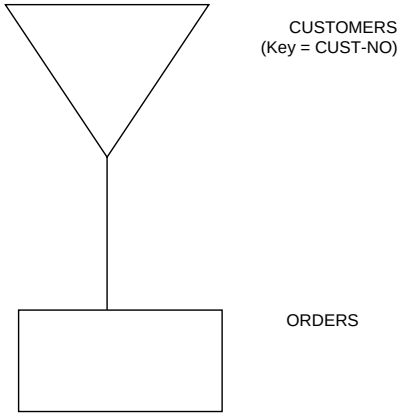
The application software package that the user has installed uses a two-step process to delete customers. The end user indicates via a transaction screen those customers they wish to delete. The result of this transaction is a transaction log file which contains the customer number of those customers "tagged" to be deleted. That night, a batch job runs and deletes the customer and their corresponding orders from the database.

The problem is the end user would prefer that the customers be moved into a history database instead of being deleted.

The structure of the database is the same as the one described in our first example and is detailed below.

Database to Database Examples

Problem #1



The ideal solution would be to use the transaction file, which contains the customer to be "archived", as our driver file. The script below uses the driver file technique.

Script Explanation

```
1>  OPEN CUTS IMAGE "CUST PASS 1"
2>  OPEN DRIVER FIXED DRIVER
3>  OPEN HIST IMAGE CUSTH &
4>   pass=PASS mode=1
5>  FORMAT RECORD FLAT_FMT
    1->  CUST-NO : X12
    13-> END
8>  HEADER "OE0900J", $TAB 120, $PAGENO
9>  HEADER $CENTER, "Order Entry"
10> HEADER $CENTER, "Customer Archival"

11> HEADER $CENTER, $TODAY
12> HEADER
13> READ CUST-NUMBERS = DRIVER &
14>  FORMAT FLAT_FMT
15>   READ CUST-IN-DB = CUST.CUSTOMERS &
16>    FOR CUST-NO = &
17>    CUST-NUMBERS.CUST-NO
18>     COPY CUST-IN-DB TO HIST.CUSTOMERS
19>     PRINT "Customer:",CUST-NO
20>     READ O = CUST.ORDERS FOR &
21>      CUST-NO = CUST-IN-DB.CUST-NO
22>       COPY O TO HIST.ORDERS
```

```
23>        PRINT $TAB 5,ORDER-NO,
24>        PRINT ORDER-DESC
25>        DELETE O
26>      ENDREAD
27>      DELETE CUST-IN-DB
28>    ENDREAD
29> ENDREAD
30> GO
```

In line 1, Warehouse opens the IMAGE CUST database using mode 1 and the password PASS and tags it CUST. In line 2, Warehouse opens the fixed file DRIVER and tags it DRIVER. In lines 3, and 4, Warehouse opens the IMAGE database HISTDB using mode 1 and the password PASS and tags it HIST.

In line 5, the record definition of DRIVER is supplied via the FORMAT statement. DRIVER contains the CUST-NO field which contains 12 bytes of data. The end indicates to Warehouse that you are done defining fields in the format statement.

In lines 8 through 12, the header which is printed on the top of every page is defined. This header statement doesn't belong to a particular report (i.e. one defined using an open statement), but the standard report which is printed with every Warehouse run. Report headers are defined using the HEADER statement. $TODAY contains the today's date and $PAGENO contains the current page number. Notice the use of $CENTER to center the text on the page. A HEADER statement with no other fields generates a blank line.

The selection of records to be archived has already taken place in the application package. Key values of the selected records are contained in the file DRIVER. All that is necessary is to read the contents of the DRIVER file and archive based on the contents of that file.

In the read loop tagged, CUST-NUMBERS, on lines 13 and 14, Warehouse reads the file tagged DRIVER

serially using the format described in the FORMAT tagged FLAT_FMT.

In lines 15 through 17, the read loop tagged CUST-IN-DB reads the CUSTOMERS dataset matching on the CUST-NO in the CUST-NUMBERS read loop. In line18, the selected customers are copied to the CUSTOMERS dataset in the database tagged HIST. Line 19 prints a line showing the customer number.

Lines 20 and 21 define the read loop tagged O. In this read loop, Warehouse reads the ORDERS dataset in the database tagged CUST matching on the CUST-NO in the read loop tagged CUST-IN-DB. In line 22, the selected orders are copied to the ORDERS dataset in the database tagged HIST. Notice that a report is being generated during the archival process. The report could be generated at a later time, but it is easier to do it while the archival process is occurring. In line 24, Warehouse deletes the order.

Finally in line 27, Warehouse deletes the customer.

Comments

A common problem is knowing when to use a `format` statement and when to use a `define` statement. The `format` statement simply describes the format of existing data. The `define` statement creates a local variable with the record layout that you describe. So, if you need a place to keep data until you are ready to write it out, use a `define`. If you need to describe the layout of data that already exists, use a `format`.

Producing reports can be accomplished by either using the standard output, like in this example, or by using the OPEN statement and opening up a REPORT type file. If you elect to use the OPEN statement method, you can create as many reports as your script requires. To write to a particular report, print to that report tag, e.g. `print [exprpt] fieldname, field2`. For more information on using this method, see REPORT in the chapter entitled, **Data File Types** in the *Warehouse*

Reference Manual under REPORT file type.

Using the system constants, like $PAGENO and $TODAY, can make life simpler. These constants are described in the chapter **Expressions** in the *Warehouse Reference Manual*. The dates can be manipulated to be printed in different formats by using the date functions also described in that same chapter.

The use of other files to drive a Warehouse script is common. It is sometimes used as in this example, because another process has already "pre-selected" our data for us. Or maybe another process, like SUPRTOOL, is faster, because of the data structures, at selecting the key values for us to process. Whatever the reason you choose to use a driver file, Warehouse is able to integrate files from other sources. This technique is applicable for all types of projects.

Database to Database Examples

Script Type

Maintaining logical data structures when moving from database to database.

Applicability

Universal technique which could be used in test environments, data conversion or creation of historical environments.

Technique Illustrated

Creating a test environment where logical data structures must be maintained as we move information from the production environment to the test environment. IMAGE to IMAGE.

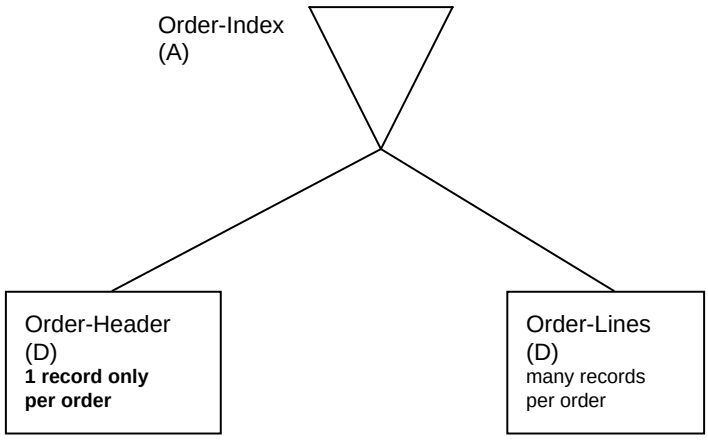
Background

This example illustrates how to maintain data integrity when using a detail dataset as a pseudo manual master. Sometimes it is useful to use a detail dataset to store information which truly belongs in a manual master dataset.

For example, it may be useful to have several automatic masters connected to order header information. However, it is not possible to have the order header information in a manual master because IMAGE does not allow manual masters to be associated with automatic masters. The alternative is to put the order header information in a detail dataset. However, the disadvantage to using this technique is that the application is then responsible for ensuring that one and only one order header record exists per order.

The following example creates a test database using Warehouse with a data structure like this. Warehouse must check to see if an order header exists prior to adding an order header record.

The data structure below is the structure used in the solution presented.



Notice that the detail dataset ORDER-HEADER is acting as a pseudo master.

Script Explanation

```
1>  OPEN ORD IMAGE ORDER pass=R mode=5
2>  OPEN TEST IMAGE TESTDB pass=W mode=3
3>  DEFINE COUNTER : I1
4>  DEFINE THERE : X1
5>  SETVAR COUNTER = 1
6>  READ OI = ORD.ORDER-INDEX &
7>   FOR COUNTER <= 1000
8>    READ OH = ORD.ORDER-HEADER &
9>     FOR ORDER-NO = OI.ORDER-NO
10>     SETVAR THERE = 'N'
11>     read chkoh = test.order-header &
12>      FOR ORDER-NO = OH.ORDER-NO &
13>      AND THERE = 'N'
14>       SETVAR THERE = 'Y'
15>     ENDREAD
16>   IF THERE = 'N' THEN
17>     COPY OH TO TEST.ORDER-HEADER
18>     READ LINES = ORD.ORDER-LINES &
19>      FOR ORDER-NO = &
20>      OI.ORDER-NO
21>       COPY LINES TO TEST.ORDER-LINES
22>     ENDREAD
23>   ENDIF
24>   ENDREAD
25>   SETVAR COUNTER = COUNTER + 1
26> ENDREAD
27> GO
```

Warehouse opens the database ORDER using mode 5 and the password READ and tags it ORD in line 1. Warehouse opens the database TESTDB using mode 3 and the password WRITE and tags it TEST in line 2.

Next in line 3 the variable COUNTER is defined. COUNTER is used to extract 1000 orders. COUNTER is initialized to 1 in line 5. The variable THERE is defined, in line 4, as a one character alphanumeric field.

The read loop tagged OI, defined in lines 6 and 7, reads the automatic master dataset, ORDER-INDEX, in the database tagged ORD for 1000 orders. By reading the automatic master, Warehouse is able to access the detail datasets by key value. This is preferable to reading one of the detail sets serially and then proceeding to the other datasets by key value.

The read loop tagged OH, defined in lines 9 and 10, reads the ORDER-HEADER dataset in the database tagged ORD matching on the ORDER-NO contained in both datasets. The variable THERE is set to N in line 10.

In the read loop tagged CHECKOH, defined in lines 11 through 13, Warehouse reads from the target file to ensure that we have only one order header record. The read loop reads from the ORDER-HEADER dataset in the database tagged TEST matching on ORDER-NO from ORDER-NO in the read loop tagged OH and checks that the value in the variable is N. This technique of reading the target file can be very handy in maintaining logical and referential data integrity. If the a record is found, Warehouse executes the statements within the read loop which causes our switch to be changed to Y. If no record is found, the switch's value remains N.

If the value of THERE is N then Warehouse copies

the selected OH records to the ORDER-HEADER dataset in the database tagged TEST in line 17. In the read loop tagged LINES defined in lines 18 and 19, the corresponding records in the dataset ORDER-LINES in the database tagged ORD are read and copied to the test database. The counter is incremented and then the next order is processed in line 25.

Comments

Here is another example of checking the target file before processing any further to decide what should happen.

Database to Database Examples

Script Type

Checking records counts before moving data.

Applicability

Checking data volumes is common when the receiving target may need to be expanded.

Technique Illustrated

Checking record counts prior to creating the test environment. IMAGE example.

Background

It is desirable to create test environments which are a subset of the production environment. These environments can be used for a number of reasons including: duplicating problems, developing enhancements, and training end users.

Knowing that you want 10% percent of your customers out of the order entry system for your test environment doesn't give a good idea on how to size your test environment.

The following example shows how determine the correct size of your test environment before copying information into it.

Script Explanation

```
1>  OPEN PROD IMAGE PROD pass=R mode=5
2>  read m = prod.m for selection = "Y"
3>    read d = prod.d for key = m.key
4>    endread
5>  endread
6>  go
```

Warehouse opens the database `PROD` using mode `5` and the password `READ` and tags it `PROD` in line 1.

The read loop tagged `m`, defined in line 2, reads the records in the dataset m for the records that have Y as the value in the data item selection.

The indented read loop d, defined in line 3, reads associated records in the dataset d.

At the end of the execution of this script, statistics are printed. The statistics tell you how many records fulfilled your selection criteria for each of the datasets read. Use these numbers to size your test database.

Comments

In environments were disc space is limited, this is a handy technique to use only the space needed.

Database to Database Examples

Script Type

Handling data conversion issues including logical data conversion changes.

Applicability

Although this technique is more common in a data conversion project, you might have a need for it in creating test environments.

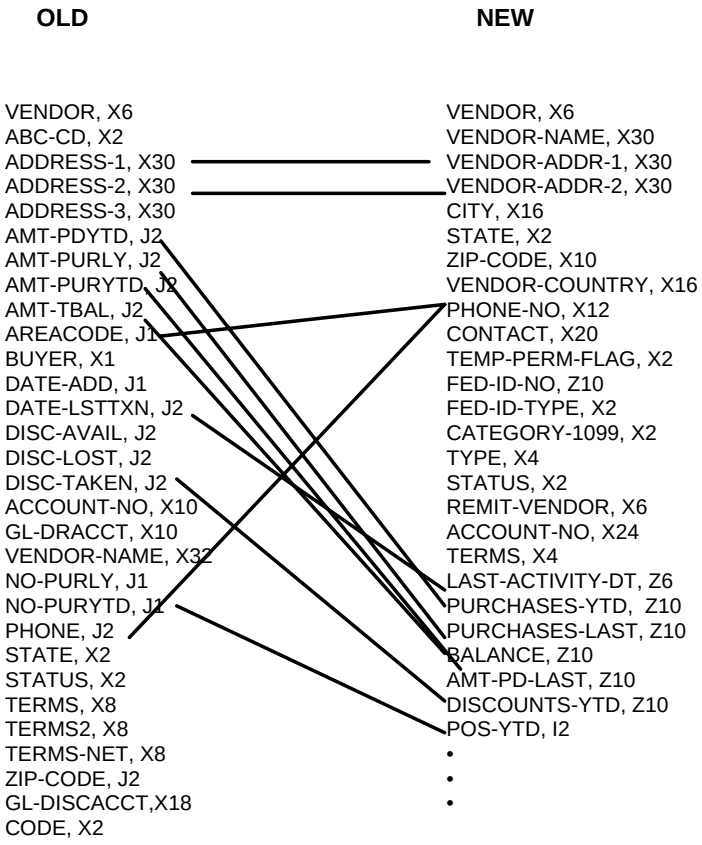
Technique Illustrated

Conversion example. Illustrating data structure conversion with the same data file type. IMAGE to IMAGE.

Background

Illustrates Warehouse's strong data conversion features. In this particular example, the user is converting from one application system to another. The data being collected is, for the most part, the same. The example illustrates how to deal with: name mismatches, data type conversions, data merging, and data mapping changes. This technique can be used for any data movement between the same or different file types. For this example, we are moving data from one IMAGE database to another with very different structural attributes.

The user is changing accounts payable systems and wants to move the vendors in the old accounts payable system to the new accounts payable system. The two vendor datasets are depicted below. The lines indicate the relationships between the different data items in the each of the datasets.



The key to understanding this example is remembering how Warehouse handles structural differences. Warehouse copies data based on item name. Those items which have the same names are handled automatically. This includes:

- data mapping, i.e. change in the location of a particular data item
- data type conversion, e.g. an X field changing to a Z field
- data length changes, e.g. an I type field used to be one character long and now it is defined as an I2.

Warehouse does not copy fields that do not have a corresponding data item in the receiving dataset. Warehouse initializes any data items which do not have a match with either spaces or zeroes depending on data type.

Our script moves the old record into a record variable. It then uses the setvar statement to handle data items that do not have the same name.

It could also use variables to initialize new data items with static values if that was necessary or appropriate.

Script Explanation

```
1> OPEN OLD IMAGE OLDDB pass=R mode=5
2> OPEN NEW IMAGE NEWDB pass=W mode=1
3>
4> DEFINE VEND : USING NEW.VEND
5>
6> READ V = OLD.VENDORS
7>   SETVAR VEND = V
8>   SETVAR VEND.BALANCE = AMT-TBAL + &
9>    AMT-PURYTD
10>  SETVAR VEND.PHONE = &
11>   STRING(AREACODE) &
12>   + "/" + STRING(PHONE)
13>  SETVAR VEND.VENDOR-ADDR-1 &
14>   = ADDRESS-1
15>  SETVAR VEND.VENDOR-ADDR-2 = &
16>   ADDRESS-1
17>  SETVAR VEND.PURCHASES-LAST = &
18>   AMT-PURLY
19>  SETVAR VEND.PURCHASES-YTD = &
20>   AMT-PDYTD
21>  SETVAR VEND.POS-YTD = NO-PURYTD
22>  SETVAR VEND.DISCOUNTS-YTD = &
23>   DISC-TAKEN
24>  SETVAR VEND.LAST-ACTIVITY-DT = &
25>   DATE-LASTTXN
26>  COPY VEND TO NEW.VENDOR
27> ENDREAD
28> GO
```

Warehouse opens the two databases in lines 1 and 2.

Line 3 defines a record variable which looks like the layout of the VENDOR dataset in the database tagged NEW.

Line 6 defines the read loop, V. It reads the VENDORS dataset in the database tagged OLD. Notice that there is no FOR clause. This causes all

records in the dataset to be read.

The selected record from the read loop `V` is moved to the `VEND` local variable in line 7.

Lines 8 and 9 calculated the `BALANCE` item by adding `AMT-TBAL` and `AMT-PURYTD`.

Lines 10 through 12 show an example using both string concatenation and changing data types using the `string` function. In this case, `areacode` and `phone` are numeric data types. They are changed to string types by using the `string` function.

Lines 13 through 25 are examples of dealing with a data name mismatch.

Once the record has been reconstructed in the local variable, line 26 copies the contents of the variable out to the new database.

Comments

This example illustrates a couple of very important concepts. The first principle illustrated reminds us how Warehouse moves data from source to target. We see in the example that all data items with the same name are moved, converted and transformed automatically.

For those fields which require extra work, e.g. phone number field, we can use the functions described in the **Expressions** chapter in the *Warehouse Reference Manual* to help us make the necessary transformations. Our last group of conversions is moving data from fields of different names. Any data transformation necessary to move the old field to the new field happens automatically.

The last technique of note which is helpful here, as well as when you need to take a single source and create multiple tables or the opposite - take a multiple data sources and create a single target, is the use of a local record variable. This script uses

the local record variable, VEND, as a holding area while the new record is being built. Once the new record has been built, the local variable is copied to the database.

Database to Database Examples

Script Type

Using SQLNET to access a remote Oracle database.

Applicability

Through the use of SQLNET, Warehouse is able to access remote Oracle databases on platforms not yet supported by Warehouse directly or were Warehouse licenses have not been purchased. SQLNET causes Warehouse to believe that the Oracle instance you have requested resides on the machine where the script is running.

Technique Illustrated

Accessing remote Oracle databases without using Warehouse server technology REMOTE ORACLE DATABASE ACCESS.

Background

Normally, Warehouse access remote files through its own client server technology. However, from time to time, it is necessary to access an Oracle database on system where Warehouse does not reside. If SQLNET has been installed and the databases are accessible via the machine Warehouse is executing on, Warehouse will be able to access the Oracle database.

Before trying to access the Oracle database via Warehouse, ensure that the connection is working through SQLNET by:

1. Issue the appropriate values for the Oracle SID and Oracle HOME.
2. Run SQLPLus and access the alias, e.g. `SQLPLUS scott/tiger@alias`.
3. Verify the instance by checking SQLNET_CHECK, e.g. DESCRIBE SQLNET_CHECK. There should be at least column listed, e.g. CONNECTED_TO_ALIAS.

Once this is complete and all looks well, access the Oracle database can be accomplished like in the example below.

Database to Database Examples

Script Explanation

```
1> open remote oracle scott/tiger@prod
2> read cust = remote.cust_table
3>   print 'Customers from production',
4>   print cust_id
5> endread
6> go
```

Warehouse opens the remote Oracle database in line1. Notice the alias name is indicated after the Oracle user and Oracle password are given. When executing this script, Oracle will indicate if the connection has been made successfully or not. All the usual Oracle security will occur.

Line 2 reads the `cust_table` table from the remote Oracle database. Notice no selection criteria is provided. This will result in all rows being read from this table.

Line 3 prints the string, `Customers from production.`

On the same print started in line 3, the `cust_id` is printed.

Line 5 terminates the read block.

Line 6 begins the execution of script. No data will be read until this command is entered.

Comments

If you are going to be accessing any Oracle database, you need to run the version of Warehouse that permits Oracle access. See the *Warehouse Reference Manual* for more information.

Freeze File Examples

Script Type

Moving historical data to an offline freeze file.

Applicability

Freeze files could be used in populating a data warehouse to "freeze" a particular moment in time. They can also be used to freeze test data before destructive testing is done so that you can restore the test database back to the state it was in before the destructive test.

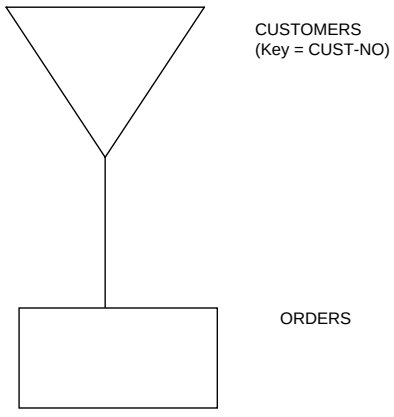
Technique Illustrated

Illustrates moving customers offline only if all of the orders are complete. Also introduces printing and variables. IMAGE to freeze file.

Background

The user has a manual master called CUSTOMERS that contains information about every customer. Linked to the CUSTOMERS dataset is the ORDERS dataset. The ORDERS dataset contains information about every order placed. The data item linking these two datasets is called CUST-NO.

Problem #1



The user wishes to archive customers and orders for customers who have not been active in one year and whose orders are complete.

To do this, two passes are made through the ORDERS chain. One pass is to determine if all of the orders are complete. The second pass is

necessary to archive and delete the orders.

In addition to this archival task, the user also wishes to generate a report of all of the archived orders. The report should display the customer number, order number, order description and order date, and be sorted by customer number and order number.

A couple of approaches can be used with this situation. The technique that is being used here is "re-reading" the data when all the conditions have been met. So the first time, we read the order lines to check and see if they are all complete. If they are not, we stop processing the record immediately. If they are all complete, we "re-read" the data for the copy and delete process.

Script Explanation

```
 1> open cust image cust.data pass=WRITER &
 2> mode=1
 3> create ordera archive ordera
 4> header $tab 40, &
 5>  'Archived orders as of', $today
 6> header
 7> header 'Cust No Order No    Desc', &
 8>  $tab 40, 'Date'
 9> define all-complete :x1
10>
11> read customers = cust.customers for &
12>  last-active < $today - 19010000
13>   setvar all-complete = 'y'
14>   read chk-order = cust.orders for &
15>    cust-no = customers.cust-no &
16>    and all-complete = 'y'
17>     if order-status <> "CMPL" then
18>       setvar all-complete = 'n'
19>     endif
20>   endread
21>   if all-complete  = 'y' then
22>     copy customers to ordera.cust
23>     read orders = cust.orders for &
24>      cust-no = customers.cust-no &
25>      order by cust-no, order-no
```

```
26>       copy orders to ordera.orders
27>       print cust-no, order-no,
28>       print order-des:20,order-date
29>       delete orders
30>     endread
31>   delete customers
32>   endif
33> endread
34> go
```

The `open` statement, which accesses files which already exist, is used in line 1 to open the IMAGE database `cust.data` and the database is tagged `cust`.

The `create` statement, which accesses files which do not exist, is used to create an archive called `ordera` and tagged `ordera`.

Lines 4 and 5 create a header line. Notice the use of the `$tab` to indicate the placement of the string and the use of the `$today` to print today's date. The format of `$today` can be controlled with the date functions. The default format of `$today` is described in the **Expressions** chapter of the *Warehouse Reference Manual.*

Line 6 results in a blank header line.

Lines 7 and 8 generate a header line which is used to print field name headers. Notice the use of the `&` to indicate the continuation of the header statement beyond one line.

Line 9 defines a local variable for use in our script.

Lines 11 and 12 define the read loop tagged, `customers`. The read loop selects records out of the `customers` dataset in the database tagged `cust` which have a `last-active` date less than one year ago. Notice that because `$today` is a numeric value arithmetic can be done.

Line 13 initializes the `all-complete` variable.

Lines 14 and 15 define the read loop tagged `chk-order`. This read loop selects those records from the `orders` dataset in the database tagged `cust` whose `cust-no` matches the one selected in the `customers` read loop and continues until the local variable's, `all-complete`, value is `y`.

Line 17 sets up a block of conditional processing for those records which are not `CMPL`. This block is terminated by the `endif` statement in line 19.

Line 18 sets the `all-complete` variable to `n`.

Line 20 terminates the `order` read loop.

Line 21 begins an `IF` block. If the local variable, `all-complete`, is set to yes, then the statements within the block are executed. This `IF` block is terminated by the `endif` on line 32.

Line 22 copies the records selected in the `customers` read loop to the file tagged `ordera` and given the name `cust`.

Comments

For more information on date handling, see **Expressions** in the *Warehouse Technical Manual*.

All archive tapes can be read on any of the platforms supported by Warehouse.

Freeze File Examples

Script Type

Moving historical data to an offline freeze file.

Applicability

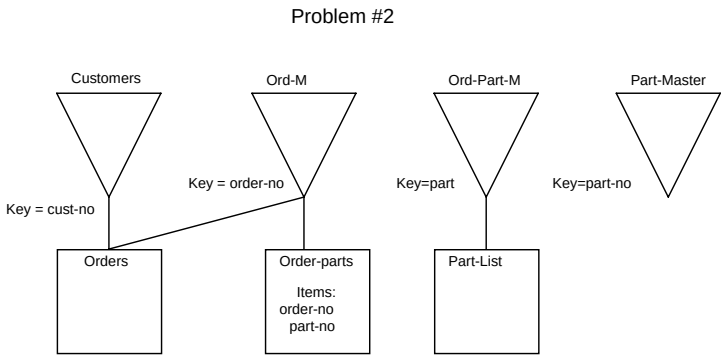
Managing historical data is an universal problem that spans both relational and hierarchical databases. The problem first presents itself as slow performance and gets worse from there. Managing historical information must be a part of every application's routine maintenance.

Technique Illustrated

Illustrates an example which archives and deletes selected orders and all associated information. The script follows the order through many logical and physical chains in four datasets. Also illustrated are date handling techniques and functions. Freeze file and IMAGE example.

Background

The next example is an order entry system. The database structure of the data to be archived is shown below.



The ORDERS dataset and ORDER-PARTS dataset are logically related via the ORDER-NO field. The ORDERNO field and PART-NO field are concatenated to create the key PART for the ORD-PART-M and the PART-LIST datasets. The PART-MASTER contains a description of the part.

The user wishes to archive all "old" orders, i.e. closed orders that are six months or older, for those customers that don't have a current order open. The CUSTOMERS dataset has a status field

which has the value OPEN if there are open orders associated with the customer. The ORDERS, ORDER-PARTS, and PART-LIST need to be archived and deleted from the database. The user also wants a report of all parts which were archived.

Script Explanation

```
 1> open cdb image "custdb WRITE 1"
 2> create custarc archive tape
 3> define part-order : x18
 4> define prt-desc : x80
 5> header 'Part', $tab 20, 'Desc.', &
 6>  $tab 60, 'Qty'
 7> header
 8>
 9> read c = cdb.customer &
10>  for cust-status <> "OPEN"
11>   read orders = cdb.orders for &
12>    cust-no = c.cust-no and &
13>    order-status = "CMPL" and &
14>    order-date < &
15>    yyyymmdd(daynum($today) - 180) &
16>    - 19000000
17>     copy orders to custarc.orders
18>     read op = cdb.order-parts for &
19>      order-no = orders.order-no
20>       copy op to custarc.order-parts
21>       setvar part-order = &
22>        order-no + part-no
23>       read parts =  &
24>        cdb.part-list &
25>        for part = part-order &
26>        order by part
27>         copy parts to  &
28>          custarc.part-list
29>         READ PM = CDB.PART-MASTER &
30>          FOR PART-NO = OP.PART-NO
31>           SETVAR PRT-DESC = &
32>            PART-DESC
33>         ENDREAD
34>         PRINT PART, PRT-DESC:40,
35>         PRINT $TAB 60, QTY
36>         DELETE PARTS
37>       ENDREAD
38>       DELETE OP
```

```
39>      ENDREAD
40>      DELETE ORDERS
41>    ENDREAD
42> ENDREAD
43> GO
```

Line 1 opens the IMAGE database CUSTDB and tags it CDB. Line 2 creates an archive file, TAPE, and tags it CUSTARC.

Lines 3 and 4 define two local variables.

Lines 5 and 6 define a header. Notice the definition of the header is split over two lines. The & character indicates that we are continuing the definition of a single header line.

Line 7 defines the second line of the header. Headers are printed at the top of each page printed.

Lines 9 and 10 define the read loop, C, which reads the dataset CUSTOMER from the database tagged CDB for those records whose status is not equal to OPEN.

Lines 11 through 16 define the read loop, ORDERS, which selects records from the ORDERS dataset whose order-status is complete and order-date is less than 6 months ago. Note the use of the DAYNUM and YYYYMMDD functions. They are used to calculate a date six months ago.

The records selected in the ORDERS read loop are copied to the archive tape in line 17.

Lines 18 and 19 read the associated records in the ORDER-PARTS dataset matching on ORDER-NO from the ORDERS read loop.

Line 20 copies the selected records from the OP read loop to the archive tape.

The local variable, PART-ORDER, is assigned the

value of the data item ORDER-NO concatenated together with the value of the PART-NO data item in line 22.

Lines 23 through 26 define the read loop PARTS, which reads the PART-LIST dataset for the matching PART-ORDERS and sorts them by PART.

Those selected records are then copied out to the archive tape in lines 27 and 28.

In lines 29 through 31, the PM read loop reads the part-master dataset for records matching on the part number from the OP loop.

The description field is saved in the local variable, PART-DESC in lines 31 and 32.

Lines 34 and 35 generate a single print line. The comma at the end of line 34 indicate the definition of the print continues on the next line.

Comments

Note the use of upper and lower case. Warehouse is not case sensitive. You can use upper and lower case as you desire. Studies have shown that lower case is easier to read.

Freeze File Examples

Script Type

Creating a historical archival log and moving data to a offline freeze file.

Applicability

Many times when moving information, it is nice to create a log file of the transactions that have been moved. This example could be applied to EDI, data conversion or synchronization applications.

Technique Illustrated

Copying to an archive file and an output file at the same time. Illustrates creating a small database containing selected items of data which have been archived. Freeze file and IMAGE example.

Background

The user would like to be able to know when the information was archived in order to mount the right archive tapes to retrieve the information. In order to be able to do this, the user is going to create a small database which contains the patient number, name, and date archived. During the archive process, Warehouse copies the data to the archive tape and copies the selected data items to the archive log database.

Script Explanation

```
1>  OPEN PATIENT IMAGE "PATDB WRITE 1"
2>  CREATE PATARC ARCHIVE TAPE
3>  OPEN ARCHLOG IMAGE LOGDB PASS=W &
4>  MODE=1
5>  * LOGDB contains the patient number,
6>  *  patient name, date archived
7>  DEFINE PATLOG : USING &
8>   ARCHLOG.PAT-ARCHIVED
9>
10> READ PE = PATIENT.PAT-ENCOUNTER &
11>  FOR STATUS = "I"
12>   COPY PE TO PATARC.PAT-ENCOUNTER
13>   SETVAR PATLOG = PE
14>   SETVAR PATLOG.DATE-ARCHIVED = &
15>    $TODAY
16>   COPY PATLOG TO ARCHLOG.PAT-
ARCHIVED
17> ENDREAD
```

```
18> GO
```

In line 1, Warehouse opens the database PATDB using mode 1 and the password WRITE and tags it PATIENT. In line 2, Warehouse creates the archive file TAPE and tags it PATARC. In line 3, Warehouse opens the database LOGDB using mode 1 and the password W.

Lines 5 and 6 are comments. All comment lines begin with *. All text after the * is ignored.

Lines 7 and 8 define a variable, PATLOG, using the same layout as the dataset PAT-ARCHIVED in the database tagged ARCHLOG.

The read loop tagged, PE, in lines 10 and 11, reads the PAT-ENCOUNTER dataset in the database tagged PATIENT for all patients which are inactive. These selected records are copied, in line 12, to the archive file tagged PATARC and named PAT-ENCOUNTER. In line 13, the SETVAR statement copies only the data items from PAT-ENCOUNTER whose name matches those in the variable PATLOG. Notice that we use the read tag PE to tell Warehouse to move the data into a variable. In line 14 and 15, the data item DATE-ARCHIVED is initialized to a system constant containing today's date, $TODAY. Once we are finished constructing the record, the variable PATLOG is copied, in line 16, to the PAT-ARCHIVED dataset in the database tagged ARCHLOG.

Comments

Having a file which contains part of the information that was archived is used frequently when large amounts of detailed information are generated by the application. These types of applications do not have the luxury of being able to keep three years or more of data online to help resolve customer service issues.

The files containing this subset of archived data do not necessarily need to reside on the same machine

as the production data. Many customers elect to put this information on CD on their PCs.

Freeze File Examples

Script Type

Retrieval historical data from a freeze file.

Applicability

Freeze files can be selectively retrieved from. This is nice for both populated just a portion of a test environment or a repopulation of a new data warehouse data model.

Technique Illustrated

Retrieval example. Illustrates selective retrieval and limiting the number of detail records using a variable. Freeze file and IMAGE example.

Background

This example retrieves a particular order from an archive tape which was created in the previous archive example. The order number that the user wants to retrieve is `8802351`.

As the database administrator, we know that for every order there is only one `ORDERS` record. Warehouse is smart enough to only read the minimum number of master records, but it does read all of the detail records. Warehouse is directed to only select one `ORDERS` record through the use of a variable. Once the `ORDERS` record has been selected, all of the associated records are copied from the tape.

Script Explanation

```
 1> OPEN CUSTARC ARCHIVE TAPE
 2> OPEN CUST IMAGE CUSTDB PASS=WRITER &
 3>  MODE=1
 4> DEFINE NUM'FOUND : I1
 5> SETVAR NUM'FOUND = 0
 6>
 7> READ OLDORD = CUSTARC.ORDERS &
 8>  FOR ORDER-NO = 8602351 &
 9>  AND NUM'FOUND = 0
10>   SETVAR NUM'FOUND = NUM'FOUND + 1
11>   COPY OLDORD TO CUST.ORDERS
12>   READ OLDOP = CUSTARC.ORDER-PARTS &
13>    FOR ORDER-NO = OLDORD.ORDER-NO
14>     COPY OP TO CUST.ORDER-PARTS
15>     READ OLDPL = CUSTARC.PART-LIST &
```

```
16>       FOR PART = OLDOP.PART
17>        COPY OLDPL TO CUST.PART-LIST
18>      ENDREAD
19>    ENDREAD
20> ENDREAD
21> GO
```

Line 1 opens an existing archive file, TAPE, and tags it CUSTARC. Lines 2 and 3 open the IMAGE database, CUSTDB, using the tag CUST.

Lines 4 and 5 define and initialize a local variable, NUM'FOUND.

The read loop, OLDORD, defined in lines 7 through 9, reads those orders in the ORDERS dataset in the file tagged CUSTARC whose order number is 8602351 while the variable NUM'FOUND value is 0.

Line 10 increments our local variable.

Line 11 copies the selected record to the ORDERS dataset in the file tagged CUST.

Lines 12 and 13 read the associated parts from the dataset ORDER-PARTS in the file tagged CUSTARC for those whose order number matches the one read in the read loop tagged, OLDORD. Notice the association is made using the read tags.

Line 14 copies the selected record to the ORDER-PARTS dataset in the database tagged CUST.

Lines 15 and 16 read the selected records in the PART-LIST dataset on the archive file for those whose part number matches the one selected in the OLDOP read loop.

Line 17 copies the selected record to the PART-LIST dataset in the database tagged CUST.

Comments

Notice the use of ' in the variable name. This is allowed. The rules for variable names are covered in the *Warehouse Reference Manual*.

Another point to notice is the strength of the indented read loops. Indented read loops are not executed unless you have fulfilled the selection criteria in the "owner" loop. By using this technique, we can limit the number of records read to only those that are pertinent to our process.

Archive files must be read in the order that they were written in. To determine this order, use the `show` statement.

Freeze File Examples

Script Type

Accessing multiple freeze files.

Applicability

In audit situations, an auditor many want to see a range of accounts over many different time periods. Being able to access multiple freeze files at the same time makes this an easy and quick process.

Technique Illustrated

Retrieval example. Reporting from a database and multiple archive files. Also illustrates printing an indexed data item. Freeze file and IMAGE example.

Background

Once the data is archived, sometimes it is necessary to look at the data on more than one archive file at a time. This next example comes to us from a client. They had to archive information over many years and really never had to access it until the auditors arrived.

The auditor wanted to look at multiple years of certain account numbers on the same report. The years had been archived one year at a time onto tape. So, the user would like to produce a report using data from both the history archive files and the active production data.

Script Explanation

```
1> OPEN LASTYR ARCHIVE ORD94
2> OPEN PREVYR ARCHIVE ORD93
3> OPEN PRODDB IMAGE ORDER &
4>  pass=WRITE mode=1
5> DEFINE CURRENTAMT : I2
6> DEFINE PREVAMT : I2
7> READ ORDS94 = LASTYR.ORDERSUM &
8>  FOR PROD = "1234"
9>    SETVAR PREVAMT = QTY
10>   SETVAR CURRENTAMT = 0
11>   READ ORDS95 = PRODDB.ORDERSUM &
12>    FOR PROD = ORDS94.PROD
13>     PRINT CUST-NO,
14>     PRINT CUST-NAME
```

```
15>     PRINT "ADDRESS:"
16>     PRINT "":2, ADDR[1]
17>     PRINT "":2, ADDR[2]
18>     PRINT "":2, ADDR[3]
19>     SETVAR CURRENTAMT = QTY
20>   ENDREAD
21>   READ ORDS93 = PREVYR.ORDERSUM &
22>    FOR PROD = ORDS94.PROD
23>     PRINT "":5,"CURRENT:",
24>     PRINT CURRENTAMT &
25>      PIC "-Z,ZZZ,ZZZ.ZZ",
26>     PRINT "LAST YR", PREVAMT &
27>      PIC "-Z,ZZZ,ZZZ.ZZ",
28>     PRINT "YEAR BEFORE LAST",
29>     PRINT QTY, PIC "-Z,ZZZ,ZZZ.ZZ"
30>   ENDREAD
31> ENDREAD
32> GO
```

Warehouse opens the archive file ORD94 and tags it LASTYR in line 1. Warehouse opens the archive file ORD93 and tags it PREVYR in line 2. Warehouse opens the database ORDER using mode 1 and the password WRITE and tags it PRODDB in lines 3 and 4.

Warehouse defines two variables to hold amount fields in lines 5 and 6.

In the read loop tagged ORDS94, defined in lines 7 and 8, Warehouse reads the ORDERSUM file for PROD equal to 1234. PREVAMT is set to the value contained in the QTY field in the dataset ORDERSUM on the archive file tagged LASTYR in lines 9 and 10.

In the read loop tagged ORDS95, defined in lines 11 and 12, Warehouse reads the associated records in the ORDERSUM dataset matching on PROD. Warehouse prints a report containing customer number, name, address. Notice the syntax for referring to a specific array element. Notice the use of "": to skip a specified number of spaces. CURRENTAMT is set to the value contained in the QTY field on the database tagged PRODDB. The

PRINT statements define more than one print line. The first print line, defined in lines 13 and 14, prints the customer number and name. Lines 15 through 18 each generate a print line.

In the read loop tagged ORDS93 defined in lines 21 and 22, Warehouse reads the file ORDERSUM in the archive file tagged PREVYR matching on PROD from the read loop tagged ORDS94. Warehouse prints a report line containing order amounts from all three years. Notice the use of the PIC clause. Notice all the print statement used only generate one print.

Comments

This example introduces some basic printing concepts. The first concept is illustrated with a couple of variations in the construction of the print line. Warehouse continues a print line if: the print statement ends in a comma or it encounters an ampersand (&). If neither of these conditions exists, Warehouse generates a new print line.

The last printing concept illustrated is the use of edit masks. The edit masks put a formatting mask over the data. The various PIC clauses are detailed in the PRINT statement in the **Commands/Statements** chapter in the *Warehouse Reference Manual*.

Freeze File Examples

Script Type

Printing information from the freeze file.

Applicability

Knowing what is on the freeze file aids in retrieving the right information from it. This is true whether the file is used for populating test environments or archiving historical information.

Technique Illustrated

The most common form of "retrieval" of information is producing a report of the archived information. It is our experience that 80% of historical data access requires generating a report.

If the data resides online, companies typically use the report writers that they are using for their production reporting to produce the report of historical information. Once the data has moved off to tape, Warehouse is the method of choice.

Warehouse allows the user to produce a report of historical information directly from tape. You need not restore the information back on to the system to produce the report.

Background

The user wishes to review the sales orders which were archived in a previous script. The report demonstrates some of the more complex functions of the Warehouse reporting facility.

The report looks as follows

```
:
WED, MAY 2, 1994, 12:41 PM                                              Page:
TITLE  :  SALES ORDER LISTING

   Sales                 Date      Date       Amount       Amount
   Order     Customer  Invoiced  Regstr     Of Sales      Paid      St
----------  --------  --------  ------     --------     --------   --
 11000030    000000    860421   860421       149.72       149.72   TF
 11000050    000000    860421   860421       149.72       149.72   TF
 11000130    000000    860421   860421        29.76        29.76   TF
 11000120    000000    860421   860421         4.99         4.99   TF
 11000122    000000    860421   860421        14.28        14.28   TF
 11000130    000000    860421   860421       -14.28       -14.28   TF
 11000145    000000    860421   860421       617.70       617.40   TF
 11000530    000000    860421   860421        16.44        16.44   TF
 11000730    000000    860421   860421       -16.44       -16.44   TF
 11000722    000000    860423   860423       109.72       109.72   TF
 11000820    000000    860423   860423       121.72       121.72   TF
 11000920    000000    860423   860423       121.66       121.66   TF
 11000990    000000    860423   860423         6.97         6.97   TF
 11000999    000000    860423   860423       712.45       712.45   TF
 11001030    000000    860423   860423       204.97       204.97   TF
 11001020    000000    860425   860425          .12          .12   TF
 11002030    000000    860425   860425       187.69       187.69   TF
 11030030    000000    860425   860425        53.21        53.21   TF
•
•
•
*_____*
3001  ORDHEAD Entries Archived
*_____*
```

Freeze File Examples

Script Explanation

```
1>  INPUT AO ARCHIVE ARCORD
2>
3>  DEFINE COUNTER : I2
4>  SETVAR COUNTER = 0
5>
6>  HEADER $TODAY,",",$HOUR, $TAB 74, &
7>   "Page:", $PAGENO:4
8>  HEADER "TITLE  : SALES ORDER
LISTING"
9>  HEADER
10> HEADER $tab 4,"Sales", $tab 25, &
11>  "Date", $tab 34, "Date", $tab 44, &
12>  "Amount", $tab 55,  "Amount"
13>
14> HEADER $tab 4, "Order", $tab 12, &
15>  "Customer", $tab 22, "Invoiced", &
16>  $tab 32, "Regstr",: $tab 44, &
17>  "Of Sales", $tab 56, "Paid", &
18>  $tab 62,"St"
19> HEADER "----------", $tab 13,&
20>  "--------", $tab 23,&
21>  "--------", $tab 33, "------ ", &
22>  $tab 43,"---------", &
23>  $tab 53,"--------", $tab 62, &
24>  ""--"
25> READ OH = AO.ORDHEAD FOR &
26>  STR(SALES-ORD,1,2) = "21"
27>   READ OS = AO.ORDSHIPTO &
28>    ORDER BY SALES-ORD
29>     PRINT $TAB 2,ORDHEAD.SALES-ORD,
30>     PRINT "":4,ORDHEAD.CUSTOMER,
31>     PRINT $TAB 22,
32>     PRINT ORDHEAD.DATE-INVOICE:6,
33>     PRINT $TAB 32,
34>     PRINT ORDHEAD.DATE-REGSTR:6,
35>     PRINT $TAB 44,
36>     PRINT ORDHEAD.AMT-OFSALES:7:2,
37>     PRINT $TAB 56,
38>     PRINT ORDHEAD.AMT-PAID:7:2,
39>     PRINT $TAB 62, STATUS
40>   ENDREAD
41>   SETVAR COUNTER = COUNTER + 1
42> ENDREAD
43> PRINT
```

```
44> PRINT "*-----*
45> PRINT COUNTER,$TAB 6,
46> PRINT "ORDHEAD Entries Archived"
47> PRINT "*-----*"
48> PRINT
49> GO
```

The archive file tagged AO is opened in line 1.

Lines 2 and 3 define and initialize the local variable COUNTER.

Lines 6 through 23 define the header that appears on the top of every printed page. The header is composed of six print lines. Notice the use of data placement through $tab, the use of system constants (lines 6/7) and the generation of a blank header line (line 9).

Lines 24 through 26 define the read loop tagged, OH, which selects those records in the ORDHEAD dataset in the database AO whose SALES-ORD first two positions is equal to 21. Notice that records are sorted by SALES-ORD.

Lines 28 through 37 define a single print line. Notice the use of $tab for data placement and the use of the : to limit or describe how numeric data should be printed.

Line 39 increments the counter.

Lines 41 through 46 print lines after the data has been processed.

Comments

Reports can be put to paper, printer or files. Sometimes it is more straight-forward to "print" extract files for use with PC import or SQL loader because of its flexibility in generating numeric data with edit masks.

Freeze File Examples

Script Type

Using freeze files to create test environments.

Applicability

No matter what type of database you are using, having small logically intact test environments are a critical part of everyone's development process. Using freeze files to refresh from, whether the test environment is on the same or different machine, is helpful.

Technique Illustrated

Using archive files to "freeze" test database selection for use in destructive testing. Freeze file and Oracle example.

Background

In environments where the production environment is heavily used and inquiries against the production data slow down the online transaction process, using techniques which minimize data selection from the production environment are helpful.

A client had a number of satellite sites all using a centrally developed application. Each satellite site was sized just large enough to handle its online transaction processing demands.

However, when testing new releases of its applications liked to test it against each of the satellite site's data. The following example shows how to "save" the test data in an archive file. This archive file can be used repeatedly to "refresh" the test database.

Script Explanation

```
1>  open site1 oracle scott/tiger
2>  create s1save archive s1save
3>  read m = site1.m for selection = "Y"
4>    copy m to s1save.m
5>  endread
6>  go
```

Warehouse opens the Oracle database using user

`scott` and password `tiger`. Warehouses creates the archive file `s1save` in line2.

The read loop tagged `m`, defined in line 3, reads the records in the table m for the rows that have Y as the value in the column `selection`.

Line 4 copies the selected rows to the archive file.

Comments

The data is loaded from the archive file before each test. The test is run, data is destroyed or altered, and the test results are verified. When they are ready to test again, the tables are dropped or erased and the data is reloaded from the archive file.

Script Type

Loading data from a flat file to an Oracle database.

Applicability

Flat file to database loads are common for a multitude of applications including: outside data sources for data warehouses, EDI, synchronization application, bulk entry of application and PC data loads.

Technique Illustrated

Loading data from a flat file of multiple record types to a single table in an Oracle database. FIXED file to Oracle example.

Background

Each day a file is sent from the corporate facility with new parts added to the parts master by the engineers. At each remote site, the flat file must be converted to the right record format for addition to the Oracle table.

The following example shows how to construct a single table from multiple records from a flat file. Illustrated in this example is the use of `FORMAT` statements, record `DEFINE`s, and techniques for constructing records.

The flat file's format changes from one record type to another. All record types share a common 31 bytes which includes the record type and the part number. The remaining bytes are different for each of the record types. The "A" type records use the same format for every A record. The "B" type records, use the same format for every B record.

Script Explanation

```
 1> open f oracle scott/tiger
 2> open flat fixed flatfile
 3>
 4> *
 5> **Define the various record types
 6> *
 7> format a_rec : record
    1-> part_desc : oracle char(60)
   61-> bom_uom_cd : oracle char(12)
```

```
   73-> cubic_vol : oracle char(24)
   97-> list_price : oracle char(18)
  115-> filler : oracle char(5)
13> end
14>
15> format b_rec : record
    1-> prod_fam_number : oracle char(24)
   25-> purch_uom_cd : oracle char(12)
   37-> sales_uom_cd : oracle char(12)
   49-> shelf_life : oracle char(4)
   53-> stock_uom_cd : oracle char(12)
   65-> upc_no : oracle char(20)
   85-> vat_cd : oracle char(12)
   97-> weight : oracle char(18)
  115-> filler : oracle char(18)
25> end
26>
27> **Define the entire record including
28> * portion shared among all records
29>
30> format flatrec: record
    1-> part_num : oracle char(30)
   31-> rec_id : oracle char(1)
   32-> arec : format a_rec offset 32
   32-> brec : format b_rec offset 32
35> end
36>
37> *
38> **Target holding areas
39> *
40> define part_rec : using f.scott.part
41> define part_seg_rec : &
42>  using f.scott.part_seg
43>
44> *
45> **Records for initialization of table
46> ** files so we don't write out
47> ** last part's data
48> *
49>
50> define part_rec_init : &
51>  using f.scott.part
52> define part_seg_rec_init : &
53>  using f.scott.part_seg
54>
55> *
56> **Define flags to keep track of the
57> ** records we have seen
58> *
59> define got-a-rec : x1 value "N"
60>
61> define old-part-num : x30 value " "
62> define cur-part : &
```

```
63>  oracle number(11,0) value 0
64>
65> define lots-spaces : oracle char(100)
66> define ten-spaces : oracle char(10) &
67>  value "          "
68>
69> setvar lots-spaces = ten-spaces + &
70>  ten-spaces + ten-spaces + &
71>  ten-spaces + ten-spaces + &
72>  ten-spaces + ten-spaces + &
73>  ten-spaces +  ten-spaces + &
74>  ten-spaces
75>
76> read allrecs = flat format flatrec &
77>  order  by part_num, rec_id
78>   if part_num <> old-part-num
79>     if got-a-rec = "Y" then
80>       setvar cur-part to cur-part + 1
81>       copy part_rec to f.scott.part
82>       setvar got-a-rec = "N"
83>     endif
84>     setvar old-part-num = part_num
85>   endif
86>   if rec_id = "A" then
87>     setvar got-a-rec = "Y"
88>     if allrecs.arec.bom_uom_cd = &
89>      str(lots-spaces,1,12) then
90>      setvar allrecs.arec.bom_uom_cd &
91>       = "*"
92>     endif
93>     if allrecs.arec.part_desc = &
94>      str(lots-spaces,1,60) then
95>       setvar allrecs.arec.part_desc &
96>        = "*"
97>     endif
98>     setvar part_rec.user_part_no = &
99>      part_num
100>    setvar part_rec.part_no = &
101>     cur-part
102>    setvar part_rec.part_desc = &
103>     allrecs.arec.part_desc
104>    setvar part_rec.bom_uom_cd = &
105>     allrecs.arec.bom_uom_cd
106>    setvar part_rec.cubic_vol = &
107>     allrecs.arec.cubic_vol
108>    setvar part_rec.list_price = &
109>     allrecs.arec.list_price
110>  endif
111>  if rec_id = "B" then
112>    if allrecs.brec.purch_uom_cd = &
113>     str(lots-spaces,1,12)
114>      setvar &
115>       allrecs.brec.purch_uom_cd &
```

```
116>       = "*"
117>    endif
118>    if allrecs.brec.sales_uom_cd = &
119>     str(lots-spaces,1,12)
120>      setvar &
121>       allrecs.brec.sales_uom_cd &
122>       ="*"
123>    endif
124>    if allrecs.brec.stock_uom_cd &
125>     = str(lots-spaces,1,12)
126>       setvar &
127>       allrecs.brec.stock_uom_cd &
128>       = "*"
129>     endif
130>     setvar part_rec.prod_fam_no = 0
131>     setvar part_rec.stock_uom_cd &
132>      = allrecs.brec.stock_uom_cd
133>     setvar part_rec.sales_uom_cd &
134>      = allrecs.brec.sales_uom_cd
135>     setvar part_rec.purch_uom_cd &
136>      = allrecs.brec.purch_uom_cd
137>     setvar part_rec.shelf_life &
138>      = allrecs.brec.shelf_life
139>     setvar part_rec.upc_no = &
140>      allrecs.brec.upc_no
141>     setvar part_rec.vat_cd = &
142>      allrecs.brec.vat_cd
143>     setvar part_rec.weight = &
144>      allrecs.brec.weight
145>   endif
146> endread
147> *
148> **Handle last record
149> *
150> if got-a-rec = "Y" then
151>    copy part_rec to f.scott.part
152> endif
```

Warehouse opens the Oracle database using user `scott` and password `tiger`. Warehouses opens the fixed file `flatfile` in line2.

Lines 7 through 13 define a record format, a_rec. This record format is going to be used to define the changing part of the flat file which is different for each record type. This format describes an A record.

Lines 15 through 25 define a record format, b_rec. This format describes a B record.

Lines 30 through 35 define the shared record format of the file tagged `flat`. Notice how other formats are used to "redefine" bytes 32 through the end of the file: one for the A type records and one for the B type records.

Line 40 defines a holding area (record type variable) called `part_rec` which takes its layout from the table `scott.part` in the database tagged `f`.

Line 41 defines a record type variable, `part_seg_rec`, which takes its layout from the table `scott.part_seg` in the database tagged `f`.

Lines 50 and 51 define a record type variable, `part_rec_init`, which is used to clear out the record in between part numbers.

Lines 52 and 53 define a record type variable, `part_seg_rec_init`, which is used to clear out the record in between part numbers.

Line 59 defines a variable, `got-a-rec`, which is used as a switch to determine if we are processing an A record type.

Line 61 defines a variable, `old-part-num`, which is used to determine if we are processing a new part number.

Lines 62 and 63 defines a local variable, `cur-part`, to keep track of the assigned part number. Notice the use of the Oracle data types. Local variables can be of any of data types defined in the *Warehouse Reference Manual* in the chapter **Data Types**.

Line 65 defines a local variable, `lots-spaces`, which is used to compare against fields that are not supposed to be "NOT NULL" in the flat file.

Lines 66 and 67 defines a variable, `ten-spaces`,

which contains ten spaces. Notice the use of the `value` clause to initialize the field.

Lines 69 through 74 initialize `lots-spaces` with a hundred spaces by concatenating ten occurances of `ten-spaces`.

Lines 76 and 77 define a read loop, `allrecs`, which reads the file tagged, `flat`, using the record format `flatrec`. The file is read in sorted order. The sort order defined by the `order by` clause is part_num as the primary sort and `rec_id` as the secondary sort.

If a new part number is being processing, line 77 causes lines 79 through 84 to be executed.

If an A record is being processed, line 78 causes lines 79 through 83 to be executed.

Line 80 increments the cur-part variable by 1.

Lines 81 copies the contents of the record variable, `part_rec`, to the `scott.part` table in the database tagged `f`.

Line 82 resets the `got-a-rec` switch to `N`.

Line 84 sets the variable `old-part-num` to the current value of `part_num`.

Line 86 causes lines 87 through 110 to be executed if Warehouse is processing an A record.

Line 87 sets the switch, `got-a-record`, to `Y`.

Lines 88 through 92 check to see if the data item `bom_uom_cd` is spaces. If it contains spaces, Warehouse changes it to `*`.

Lines 93 through 97 check to see if the data item `part_desc` is spaces. If it contains spaces, Warehouse changes it to `*`.

Lines 98 through 109 move the remaining fields into the record.

If Warehouse is processing a B record, line 111 causes lines 112 through 145 to be executed.

Lines 112 through 117 check to see if the data item `purch_uom_cd` is spaces. If it contains spaces, Warehouse changes it to `*`.

Lines 118 through 123 check to see if the data item `sales_uom_cd` is spaces. If it contains spaces, Warehouse changes it to `*`.

Lines 124 through 129 checks to see if the data item `stock_uom_cd` is spaces. If it contains spaces, Warehouse changes it to `*`.

Lines 130 through 144 moves the remaining fields into the record.

Lines 150 through 152 copies the last record processed to the `scott.part` table.

Comments

This example shows how multiple records from one file can be combined in a holding area, and when all the components of the record are built, be copied to the target file.

The trick to this an example is to read the file in sorted order so that Warehouse reads all the records associated with a part together and in record type order. The IF statements control the processing for each of the record types and also handle data "problems" that appear in the source file, e.g. NOT NULL values.

Mixed File Examples

Script Type

Using a flat file to drive a Warehouse process.

Applicability

Universal.

Technique Illustrated

Illustrates using an flat file to drive an archive process. The flat file contains key values of those records which have been qualified for archiving. This flat file could have been created by another process or program such as SUPRTOOL or SQL. This technique is also useful for creating a test database containing a predetermined list of records.

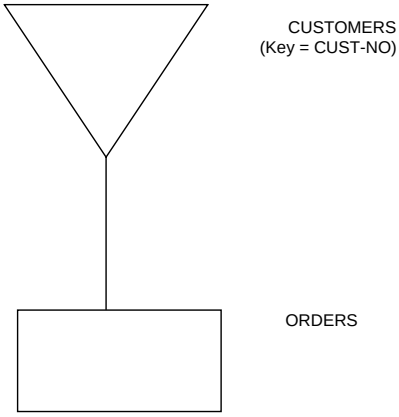
Background

The application software package that the user has installed uses a two-step process to delete customers. The end user indicates via a transaction screen those customers they wish to delete. The result of this transaction is a transaction log file which contains the customer number of those customers "tagged" to be deleted. That night, a batch job runs and deletes the customer and their corresponding orders from the database.

The problem is the end user would prefer that the customers be moved into a history database instead of being deleted.

The structure of the database is the same as the one described in our first example and is detailed below.

Problem #1



The ideal solution would be to use the transaction file, which contains the customer to be "archived", as our driver file. The script below uses the driver file technique.

Script Explanation

```
1>  OPEN CUTS IMAGE "CUST PASS 1"
2>  OPEN DRIVER FIXED DRIVER
3>  OPEN HIST IMAGE CUSTH &
4>   pass=PASS mode=1
5>  FORMAT RECORD FLAT_FMT
    1->  CUST-NO : X12
    13-> END
8>  HEADER "OE0900J", $TAB 120, $PAGENO
9>  HEADER $CENTER, "Order Entry"
10> HEADER $CENTER, "Customer Archival"

11> HEADER $CENTER, $TODAY
12> HEADER
13> READ CUST-NUMBERS = DRIVER &
14>  FORMAT FLAT_FMT
15>   READ CUST-IN-DB = CUST.CUSTOMERS &
16>    FOR CUST-NO = &
17>    CUST-NUMBERS.CUST-NO
18>     COPY CUST-IN-DB TO HIST.CUSTOMERS
19>     PRINT "Customer:",CUST-NO
20>     READ O = CUST.ORDERS FOR &
21>      CUST-NO = CUST-IN-DB.CUST-NO
22>       COPY O TO HIST.ORDERS
```

```
23>        PRINT $TAB 5,ORDER-NO,
24>        PRINT ORDER-DESC
25>        DELETE O
26>      ENDREAD
27>      DELETE CUST-IN-DB
28>    ENDREAD
29> ENDREAD
30> GO
```

In line 1, Warehouse opens the IMAGE CUST database using mode 1 and the password PASS and tags it CUST. In line 2, Warehouse opens the fixed file DRIVER and tags it DRIVER. In lines 3, and 4, Warehouse opens the IMAGE database HISTDB using mode 1 and the password PASS and tags it HIST.

In line 5, the record definition of DRIVER is supplied via the FORMAT statement. DRIVER contains the CUST-NO field which contains 12 bytes of data. The end indicates to Warehouse that you are done defining fields in the format statement.

In lines 8 through 12, the header which is printed on the top of every page is defined. This header statement doesn't belong to a particular report (i.e. one defined using an open statement), but the standard report which is printed with every Warehouse run. Report headers are defined using the HEADER statement. $TODAY contains the today's date and $PAGENO contains the current page number. Notice the use of $CENTER to center the text on the page. A HEADER statement with no other fields generates a blank line.

The selection of records to be archived has already taken place in the application package. Key values of the selected records are contained in the file DRIVER. All that is necessary is to read the contents of the DRIVER file and archive based on the contents of that file.

In the read loop tagged, CUST-NUMBERS, on lines 13 and 14, Warehouse reads the file tagged DRIVER

serially using the format described in the FORMAT tagged FLAT_FMT.

In lines 15 through 17, the read loop tagged CUST-IN-DB reads the CUSTOMERS dataset matching on the CUST-NO in the CUST-NUMBERS read loop. In line18, the selected customers are copied to the CUSTOMERS dataset in the database tagged HIST. Line 19 prints a line showing the customer number.

Lines 20 and 21 define the read loop tagged O. In this read loop, Warehouse reads the ORDERS dataset in the database tagged CUST matching on the CUST-NO in the read loop tagged CUST-IN-DB. In line 22, the selected orders are copied to the ORDERS dataset in the database tagged HIST. Notice that a report is being generated during the archival process. The report could be generated at a later time, but it is easier to do it while the archival process is occurring. In line 24, Warehouse deletes the order.

Finally in line 27, Warehouse deletes the customer.

Comments

A common problem is knowing when to use a format statement and when to use a define statement. The format statement simply describes the format of existing data. The define statement creates a local variable with the record layout that you describe. So, if you need a place to keep data until you are ready to write it out, use a define. If you need to describe the layout of data that already exists, use a format.

Producing reports can be accomplished by either using the standard output, like in this example, or by using the OPEN statement and opening up a REPORT type file. If you elect to use the OPEN statement method, you can create as many reports as your script requires. To write to a particular report, print to that report tag, e.g. print [exprpt] fieldname, field2. For more information on using this method, see REPORT in the chapter entitled, **Data File Types** in the *Warehouse*

Reference Manual under REPORT file type.

Using the system constants, like $PAGENO and $TODAY, can make life simpler. These constants are describe in the chapter **Expressions** in the *Warehouse Reference Manual*. The dates can be manipulated to be printed in different formats by using the date functions also described in that same chapter.

The use of other files to drive a Warehouse script is common. It is sometimes used, as in this example, because another process has already "pre-selected" our data for us. Another is example is to take advantage of SUPRTOOL's MR/NOBUF access against IMAGE to create a file containing just key values for Warehouse to base its processing against. Whatever your reason to use a driver file, Warehouse is able to integrate files from other sources. This technique is applicable for all types of projects.