NetBase SQL

Shadowing Handbook

QUEST SOFTWARE

610 Newport Center Drive, Suite 1400, Newport Beach, CA 92660

Quest Software
610 Newport Center Drive, Suite 1400
Newport Beach, CA 92660
USA
Tel. (714) 720-1434 / Fax. (714) 720-0426

Quest Software International, LTD
Oakmoore Court, Kingswood Road, Hampton Lovett
Droitwich, Worcs. WR90QH
UK
Tel. +44 (0) 1905 797935 / Fax. +44 (0) 1905 797936

NetBase SQL Shadowing Handbook - Revision A0195

NetBase SQL Shadowing

Table of Contents

Chapter 1
Introduction

Who should use this handbook?

The purpose of this handbook is to address the issues of customers who are interested in SQL shadowing without using any of the other NetBase features. For customers who are adding SQL shadowing to their other NetBase services, such as NFA, shadowing, spooling, or statistics, the NetBase reference manual is the appropriate text.

NetBase SQL shadowing uses a small fraction of the NetBase product. By default, when you initiate NetBase, it expects you to be using some of the other services. From this handbook, you will learn to modify those defaults, instructing NetBase to perform only SQL shadowing. This handbook will outline all of the steps necessary to fully implement SQL shadowing, giving examples of applications of all of the options.

NetBase SQL Shadowing

NetBase SQL shadowing consists of an export process and an import process which call routines from the Warm Standby Logging library in order to shadow an ALLBASE DBE. Warm Standby Logging (WSL) is a feature of SQL provided by HP. Warm Standby Logging with NetBase can be used to shadow a DBE on one node to copies on more than one node, or it may be used to shadow DBEs on multiple nodes to one consolidated copy. Also, it can be used for concurrent updating of multiple DBEs.

NetBase SQL shadowing can involve an entire DBE, or you may use it to include or exclude specific tables within a DBE. In addition, you may invoke user exits to re-route or ignore shadow information before it is passed into the network. You may read transactions out of the logfile, without sending them across to shadow machines by setting up an export user exit in a design we refer to as a "pseudo-node." Using a pseudo-node user exit, you can log all updates or deletes (or both), or perform other analysis of the DBE activity without transmitting to the remote nodes.

On the remote side, you can write user exits that ignore data, re-route data to other nodes, or do additional processing before the data is posted to the database. Finally, a user exit may be called if an error was encountered during the update of the shadow.

The key to all of this is the partition number. Updates made to a DBE by users are identified by the home partition number. Updates made to a DBE by the WSL routines will be identified by the home partition number on the originating machine. In this way two DBEs can be shadowed both ways. The export process should be configured on each machine to only send the updates made by the local users.

Chapter 2
Installation

Installation Summary

Typically, installation of the software and configuration of each system can be accomplished in less than 20 minutes per system.

NOTE: This installation process assumes that the part of WSL which is a set of routines provided by HP has been installed into an XL and that the databases to be shadowed via SQL exist.

In general, the following steps comprise the installation process:

1. Restore the NetBase software from tape.
2. Stream the installation job.
3. Configure NetBase operational parameters.
4. Define all nodes in the network.
5. Make SQLOUT, SQLIN, and TABLE entries in **NBDIR**.
6. Run **ISQL**, enabling Warm Standby Logging.
7. Run **SQLUtil**, adding additional archive logs.
8. Start SQL shadowing within **NBCTRL**.

Note: NetBase requires that the Loopback network interface must be configured and activated. Consult your NS manuals for information on the Loopback network interface.

NetBase SQL Installation

1. Restore NetBase from the Installation Tape

To create NetBase's environment and install the software, you will restore a job into PUB.SYS. This job is then streamed to create and restore the NetBase account.

Log on to MANAGER.SYS, and restore the job stream file:

```
: HELLO MANAGER.SYS,PUB
: RESTORE ;NETACCT.JOB.@;LOCAL

<<reply to the tape request>>
```

Special Note!

For users of VESOFT's STREAMX utility: To allow the NETACCT job to execute successfully if streamed using the VESOFT STREAMX utility, the following JCW must be set prior to streaming the NETACCT job:

```
: SETJCW STREAMXTEMPNONEST = 1
```

Special Note!

For users of SECURITY/3000: Prior to streaming the installation job, please enter the following command:

```
: NEWACCT NETBASE,MGR;PASS=QUEST
```

If necessary, add any required passwords to the file NETACCT.PUB.SYS.

2. Stream the Installation Job

The job will build the NETBASE account and restore the required files. Prior to streaming the job, place the tape back on-line.

```
: STREAM NETACCT
```

This job will request the tape and print a message to the console when done. If the job encounters any errors, a message telling what action is necessary will be displayed to the console.

3. Configure NetBase Operational Parameters

Minimally, NetBase must be instructed to perform SQL shadowing. This is accomplished using the **CONFIG** subsystem in the **NBCTRL** program. All configuration must be performed from the NETBASE account.

```
: HELLO MGR.NETBASE
: RUN NBCTRL
NetBase Control Program  [v.u.f]  Update n  (C) QUEST Software 1987

*> CONFIG
C> MODIFY SQL

SQL - Enable SQL Shadowing......... [N]     Y

C>
```

If this parameter is set, the control process will look for **SQLOUT** and **SQLIN** entries in the directory and start up a process for each record. The program that is run for **SQLOUT** records is **NBSQLEXP.NB**. The program that is run for **SQLIN** records is **NBSQLIMP.NB**.

4. Configure Network Nodes

For NetBase to communicate to the other systems on the network, all systems must be defined to each other. This is accomplished using the **CONFIG** subsystem in the **NBCTRL** program. All configuration must be performed from the NETBASE account.

Now, each computer in the network, including the local computer being configured, must be entered into the configuration. Some definitions:

- The **System Name** is an arbitrary name, and can be up to eight characters long. It is referenced in NetBase displays, reports and commands.
- The **Node Name** must be the fully-qualified NS node name defined in your NS network configuration. NetBase will not operate if the node name is incorrect.

To get started, select a node number for each computer being configured. Node numbers can be assigned arbitrarily, however, once a number is assigned to a particular node, that number should be used consistently for that node throughout the configuration for all nodes.

On each system, you must first define the local node number.

```
C> M GLOBAL

GLOBAL - Local Node Number......... [20]    3
GLOBAL - Error Enhancements........ [IR]
GLOBAL - Maximum Local Users....... [32]
```

Assuming three computers (named FOX, BUNNY, and SQUIRREL) are involved, on each computer, to add each node into the configuration:

```
C> ADD 3

System Name........................         FOX
Startup Access (S/I/O/A/W)......... [A]
Line Type (N/H).................... [N]
Node Name..........................         FOXTROT.QUEST.SOFTWARE
Node Connection Timeout............ [180]
```

Note: In the above example, node FOXTROT is the local node and has now been assigned node number 3 and should continue to be defined as node 3 in the node configuration on each system in the network.

```
C> ADD 1

System Name........................         SQUIRREL
Startup Access (S/I/O/A/W)......... [A]
Line Type (N/H).................... [N]
Node Name..........................         SQUIRREL.QUEST.SOFTWARE
Node Connection Timeout............ [180]
```

Note: Again, as in the previous example, node SQUIRREL should continue to be defined as node 1 throughout the configurations for each system in the network.

```
C> ADD 2

System Name........................         BUNNY
Startup Access (S/I/O/A/W)......... [A]
Line Type (N/H).................... [N]
Node Name..........................         BUNNY.QUEST.SOFTWARE
Node Connection Timeout............ [180]
```

This procedure must be repeated for each system in the network to be accessed by NetBase. The **LIST** command can be used to display the configured nodes.

```
C> L NODES

Node  Sys name   Line Type   Startup   Spool   Shadow
  1   SQUIRREL       N          N        N        N
  2   BUNNY          N          N        N        N
  3*  FOX            N          N        N        N

Currently 3 Nodes are defined
```

The asterisk (*) indicates the local node.

5. Make Directory Entries

Instruct NetBase as to which DBEs are to be shadowed. This is accomplished using the **NBDIR** program. The **SQLOUT** command is used to define which DBEs are exporting updates to which nodes. For each DBE, a separate **SQLIN** record must be entered for each node importing updates. These entries take effect with the next SQL shadowing session (started with either **START NETBASE** or **START SQL** within **NBCTRL**).

In addition to defining which DBEs are to be shadowed, you may use the **SQLIN** and **SQLOUT** commands to invoke user exits and to define pseudo nodes. User exits may be invoked prior to sending the transaction to the importing machine, prior to applying the imported transaction, or after applying a troublesome transaction. User exits are explained in detail in chapter 3, **User Exits**.

SQLOUT

Defines DBEs for outbound SQL Shadowing

SYNTAX:

```
D> SQLOUT dbename,node;HOME=home-partition[,part2[,...]]
                      [;EXECPRI={BS|CS|DS|ES}]
                      [;FREQ=wakeup-seconds]
                      [;{INCLUDE|EXCLUDE}]
                      [;USEREXIT]
                      [;RESET]
```

The node name should be specific, unless you are creating a pseudo node. In such cases, the node must be "@". This is the node to which you are sending the data.

The home partition number is the number of the local copy of the DBE.

The EXECPRI defines the execution priority for the SQL export process. The default is a linear priority of 152.

The FREQ option allows the user to specify the length of time that the export process will pause between checks to see if new records have been added to the log file. The pause only takes effect after a call to get the latest record from the log file does not find new records. The default is 20 seconds.

INCLUDE/EXCLUDE instructs NetBase to shadow or exclude the tables specified in the **TABLE** command for this database.

The USEREXIT parameter specifies that for each transaction received for the defined file, a user exit routine, named **export_user_exit**, should be called prior to the transaction being exported to the other nodes.

The RESET parameter may be used when a previous **SQLOUT** entry has become obsolete.

SQLIN

Defines DBEs for inbound SQL Shadowing

SYNTAX:

```
D> SQLIN dbename[=remname],node [;PRI=transaction-priority]
                                 [;EXECPRI={BS|CS|DS|ES}]
                                 [;USEREXIT]
                                 [;RESET]
```

The node, in this case, is the node from which data is coming.

The PRI option is used to set the priority on the BEGIN WORK for the transaction being applied. This is important in the case of deadlocks. The import process should have the highest priority of all the processes on the system, because HP resolves a deadlock by aborting the process with the lowest priority. The valid range is 0 to 255 with the highest priority going to the lowest number. The default is 0.

The EXECPRI option defines the execution priority for the SQL export process. The default is CS.

The USEREXIT parameter specifies that an import user exit routine, named either **post_sql_exit_before** or **post_sql_exit_after**, should be called with each transaction received for the defined file. (The **post_sql_exit_before** routine is applied prior to posting the imported transaction. The **post_sql_exit_after** routine is called only if an apply fails.)

The RESET parameter may be used when a previous **SQLIN** entry has become obsolete.

On the following pages are examples of the necessary **NBDIR** entries for different shadowing scenarios.

EXAMPLES:

The simplest form of SQL shadowing is to shadow a DBE to another machine. Let's say we are shadowing DBE PRODDBE from node SYSA to SYSB.

On SYSA we run **NBDIR,** and enter the following record:

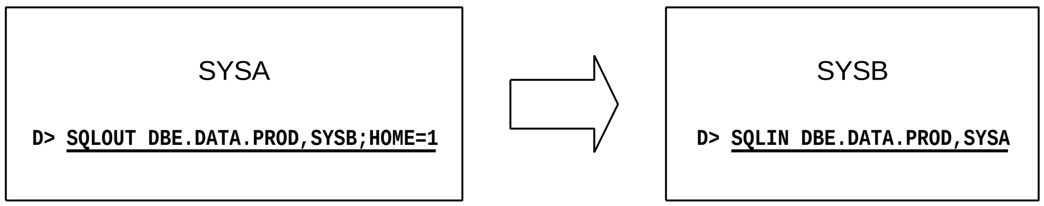
```
: RUN NBDIR.PUB.NETBASE
NetBase Directory Program  [v.u.f]  Update n  (C) QUEST Software 1987

D> SQLOUT PRODDBE.DATA.PROD,SYSB;HOME=1
```

On SYSB we enter into **NBDIR**:

```
D> SQLIN PRODDBE.DATA.PROD,SYSA
```

SINGLE DIRECTION SQL SHADOWING


SYSA
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=1
SYSB
D> SQLIN DBE.DATA.PROD,SYSA

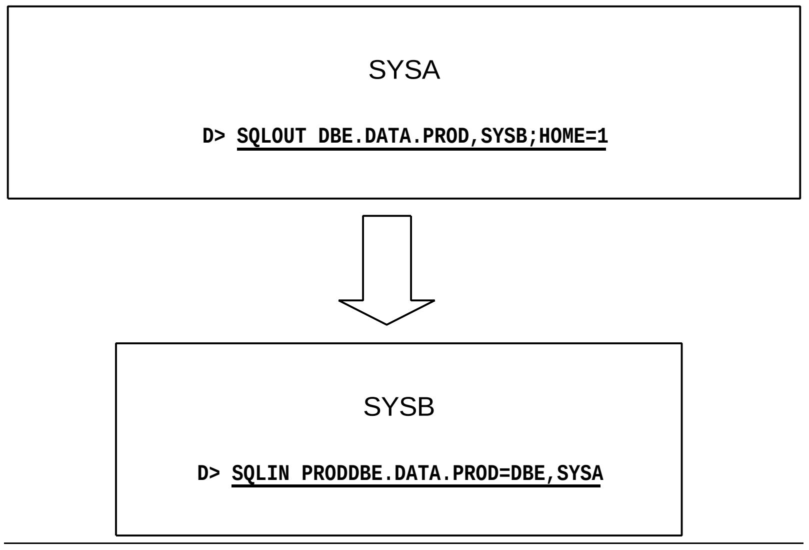
Renaming DBEs

It should be noted that the link between the **SQLOUT** and **SQLIN** matching pair is formed by the name of the DBE on the shadow (or import) side combined with the export node and the import node. If the name of the DBE on the import side is not the same as the name of the DBE on the export side, the **SQLOUT** record needs to specify the remote name. For instance, if the name of the master DBE is DBE and the name of the shadow DBE is PRODDBE, the following record should be entered on the export machine:

```
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=1
```

Note that only the file name is used, not the group or account. So, multiple entries with the same file name (with different groups and accounts) may not be used, as the directory entry only keeps the file name (and assumes the group and account). The **SQLIN** record on the import machine would be:

```
D> SQLIN PRODDBE.DATA.PROD=DBE,SYSA
```


SYSA
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=1
SYSB
D> SQLIN PRODDBE.DATA.PROD=DBE,SYSA

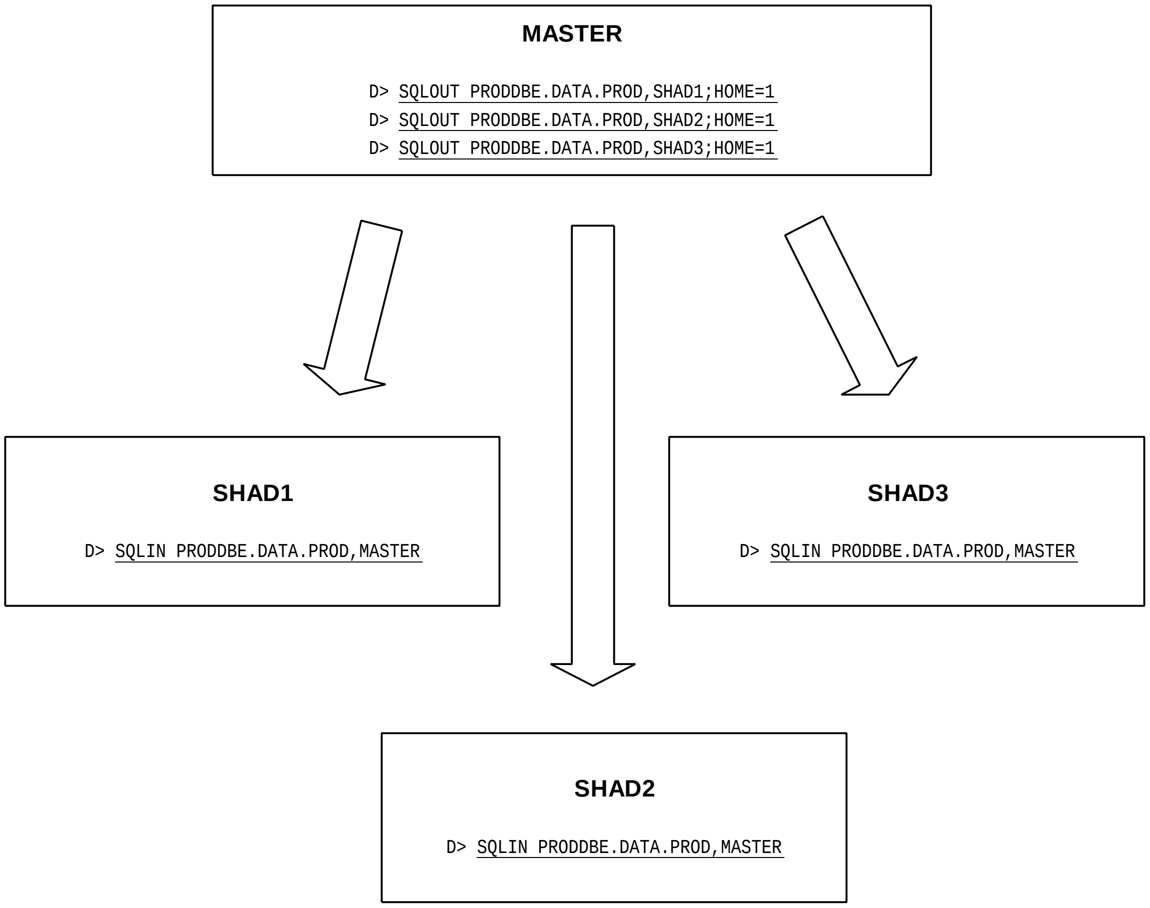
Distributed SQL Shadowing

Another way a network may be set up is to shadow a DBE from one node to a number of other nodes. If we are shadowing PRODDBE from MASTER to SHAD1, SHAD2, and SHAD3, then on MASTER, we enter:

```
D> SQLOUT PRODDBE.DATA.PROD,SHAD1;HOME=1
D> SQLOUT PRODDBE.DATA.PROD,SHAD2;HOME=1
D> SQLOUT PRODDBE.DATA.PROD,SHAD3;HOME=1
```

And on each of the other systems, we enter:

```
D> SQLIN PRODDBE.DATA.PROD,MASTER
```


MASTER
D> SQLOUT PRODDBE.DATA.PROD,SHAD1;HOME=1
D> SQLOUT PRODDBE.DATA.PROD,SHAD2;HOME=1
D> SQLOUT PRODDBE.DATA.PROD,SHAD3;HOME=1
SHAD1
D> SQLIN PRODDBE.DATA.PROD,MASTER
SHAD3
D> SQLIN PRODDBE.DATA.PROD,MASTER
SHAD2
D> SQLIN PRODDBE.DATA.PROD,MASTER

Consolidated SQL Shadowing

Or, you may want to export updates from DBEs on a number of nodes to a master (consolidated) copy on one machine. Notice that in this case our terminology involving the word "shadow" is reversed. In this set up, the DBEs on each of the export machines is unique, and the master copy is a combination of them all. For example, let's export updates to DBE from SYSA, SYSB, and SYSC to SYSD. On SYSA we enter:

```
D> SQLOUT DBE.DATA.PROD,SYSD;HOME=1
```

On SYSB:

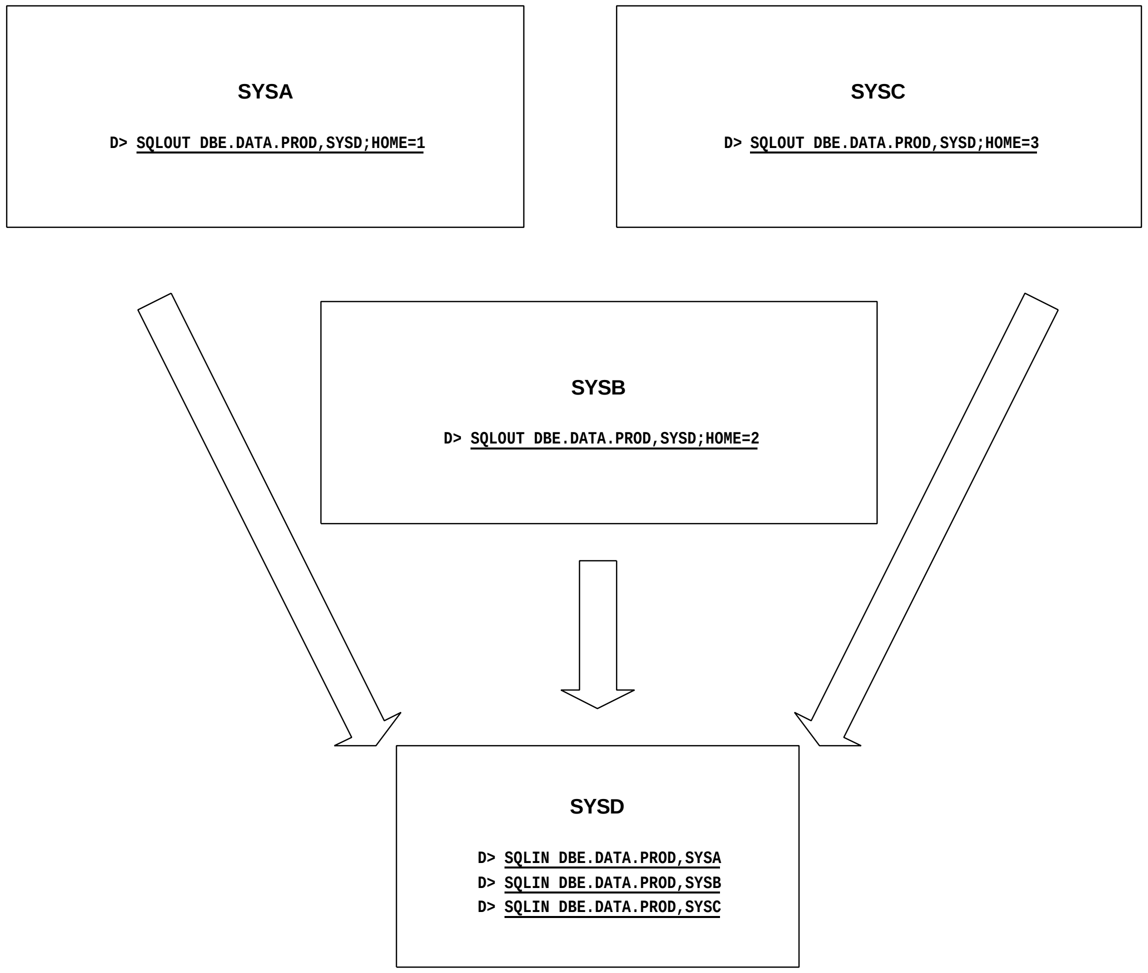
```
D> SQLOUT DBE.DATA.PROD,SYSD;HOME=2
```

On SYSC:

```
D> SQLOUT DBE.DATA.PROD,SYSD;HOME=3
```

And on SYSD:

```
D> SQLIN DBE.DATA.PROD,SYSA
D> SQLIN DBE.DATA.PROD,SYSB
D> SQLIN DBE.DATA.PROD,SYSC
```


SYSA
D> SQLOUT DBE.DATA.PROD,SYSD;HOME=1
SYSC
D> SQLOUT DBE.DATA.PROD,SYSD;HOME=3
SYSB
D> SQLOUT DBE.DATA.PROD,SYSD;HOME=2
SYSD
D> SQLIN DBE.DATA.PROD,SYSA
D> SQLIN DBE.DATA.PROD,SYSB
D> SQLIN DBE.DATA.PROD,SYSC

Multi-Directional SQL Shadowing

The most complex use of SQL shadowing is when you want to export updates from all machines in a network to all other machines in the network. To reduce the chances of synchronization problems, you should consider some form of partitioning. The idea behind partitioning is that master copies of different sections of a DBE are kept on different systems. This may be accomplished using user exits or by using **TABLE** entries in **NBDIR**.

In the example using user exits below, let's say we have PRODDBE on SYSA, SYSB and SYSC, and we want to have each node updated from the other two nodes.

On SYSA, we enter:

```
D> SQLOUT PRODDBE.DATA.PROD,SYSB;HOME=1;USEREXIT
D> SQLOUT PRODDBE.DATA.PROD,SYSC;HOME=1;USEREXIT
D> SQLIN PRODDBE.DATA.PROD,SYSB;USEREXIT
D> SQLIN PRODDBE.DATA.PROD,SYSC;USEREXIT
```

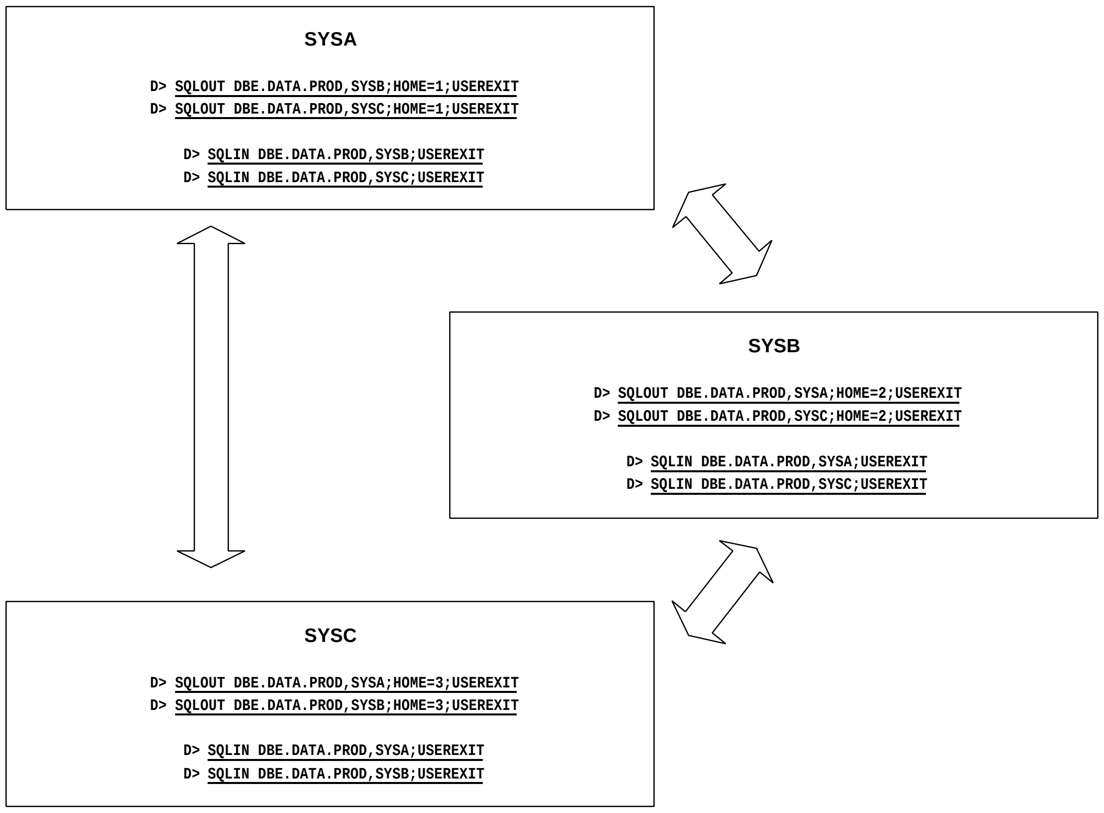
On SYSB we enter:

```
D> SQLOUT PRODDBE.DATA.PROD,SYSA;HOME=2;USEREXIT
D> SQLOUT PRODDBE.DATA.PROD,SYSC;HOME=2;USEREXIT
D> SQLIN PRODDBE.DATA.PROD,SYSA;USEREXIT
D> SQLIN PRODDBE.DATA.PROD,SYSC;USEREXIT
```

On SYSC we enter:

```
D> SQLOUT PRODDBE.DATA.PROD,SYSA;HOME=3;USEREXIT
D> SQLOUT PRODDBE.DATA.PROD,SYSB;HOME=3;USEREXIT
D> SQLIN PRODDBE.DATA.PROD,SYSA;USEREXIT
D> SQLIN PRODDBE.DATA.PROD,SYSB;USEREXIT
```

See the next page for an illustration of this example.

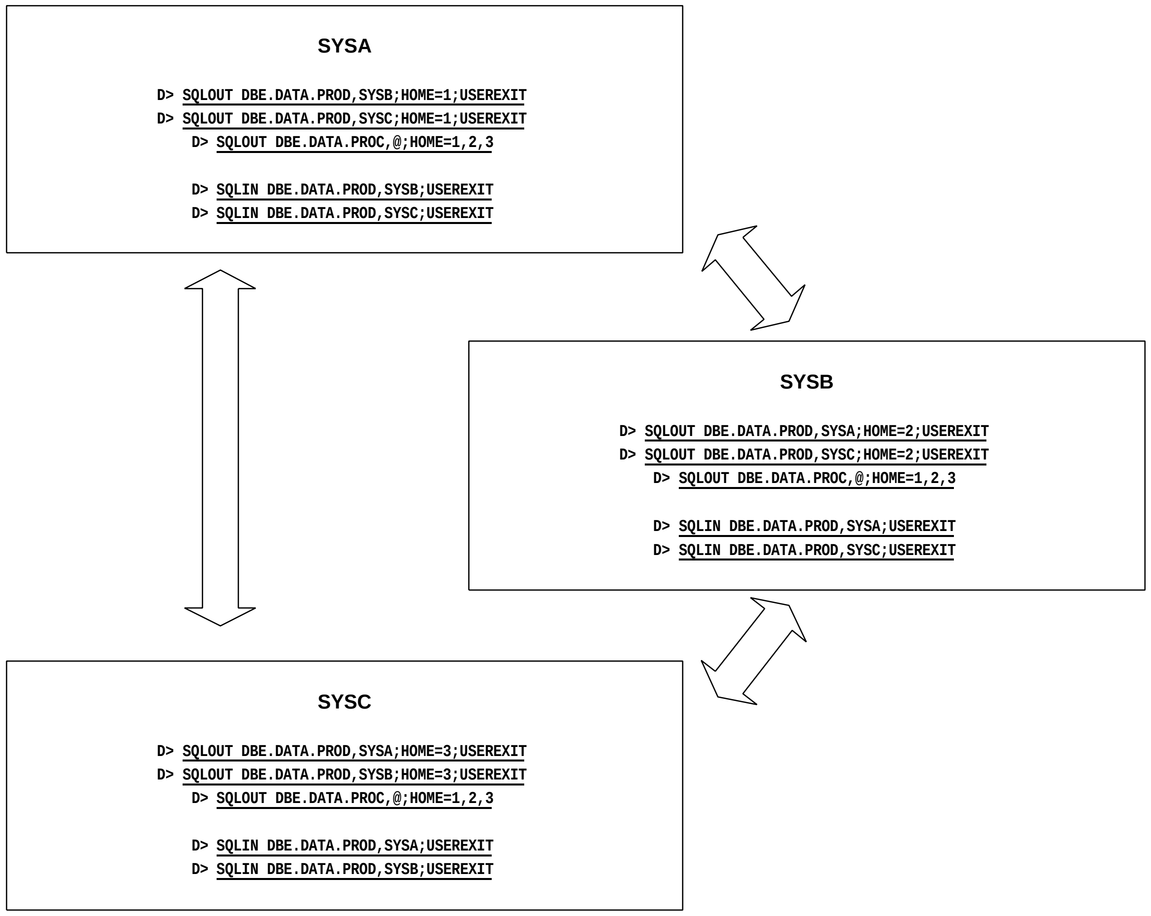

SYSA
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=1;USEREXIT
D> SQLOUT DBE.DATA.PROD,SYSC;HOME=1;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSB;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSC;USEREXIT
SYSB
D> SQLOUT DBE.DATA.PROD,SYSA;HOME=2;USEREXIT
D> SQLOUT DBE.DATA.PROD,SYSC;HOME=2;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSA;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSC;USEREXIT
SYSC
D> SQLOUT DBE.DATA.PROD,SYSA;HOME=3;USEREXIT
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=3;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSA;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSB;USEREXIT

Pseudo-Node Shadowing

If we want to have a pseudo-node process running on each of the machines, on each of the machines we would also enter the following record:

```
D> SQLOUT PRODDBE.DATA.PROD,@;PART=1,2,3
```

The user exit interprets the @ as node zero (0), and may perform whatever is defined for that node. A common application is to log all update transactions (without logging any of the rest).


SYSA

```
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=1;USEREXIT
D> SQLOUT DBE.DATA.PROD,SYSC;HOME=1;USEREXIT
D> SQLOUT DBE.DATA.PROC,@;HOME=1,2,3

D> SQLIN DBE.DATA.PROD,SYSB;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSC;USEREXIT
```

SYSB

```
D> SQLOUT DBE.DATA.PROD,SYSA;HOME=2;USEREXIT
D> SQLOUT DBE.DATA.PROD,SYSC;HOME=2;USEREXIT
D> SQLOUT DBE.DATA.PROC,@;HOME=1,2,3

D> SQLIN DBE.DATA.PROD,SYSA;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSC;USEREXIT
```

SYSC

```
D> SQLOUT DBE.DATA.PROD,SYSA;HOME=3;USEREXIT
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=3;USEREXIT
D> SQLOUT DBE.DATA.PROC,@;HOME=1,2,3

D> SQLIN DBE.DATA.PROD,SYSA;USEREXIT
D> SQLIN DBE.DATA.PROD,SYSB;USEREXIT
```

Excluding Tables

The previous examples have assumed complete shadowing of DBEs. Thanks to the INCLUDE and EXCLUDE options on the **SQLOUT** commands, you may limit the shadowing of tables or set up vertical partitioning. If you specify INCLUDE on an **SQLOUT** command, the corresponding **TABLE** records for that DBE reflect the only tables to be included in shadowing for that DBE. Conversely, if you use the EXCLUDE option on the **SQLOUT** command, the corresponding **TABLE** records reflect the tables to be omitted from shadowing.

TABLE

This command is used to enter TABLE records to be included or excluded from SQL shadowing.

SYNTAX:

```
D> TABLE dbename,node,tablename[,RESET]
```

The *tablename* entry designates the table being included/excluded (depending on the associated **SQLOUT** command) from shadowing.

The RESET option removes the table entry from the directory.

EXAMPLES:

For our example, let's assume DBE includes five tables, and we are shadowing it from SYSA to SYSB.

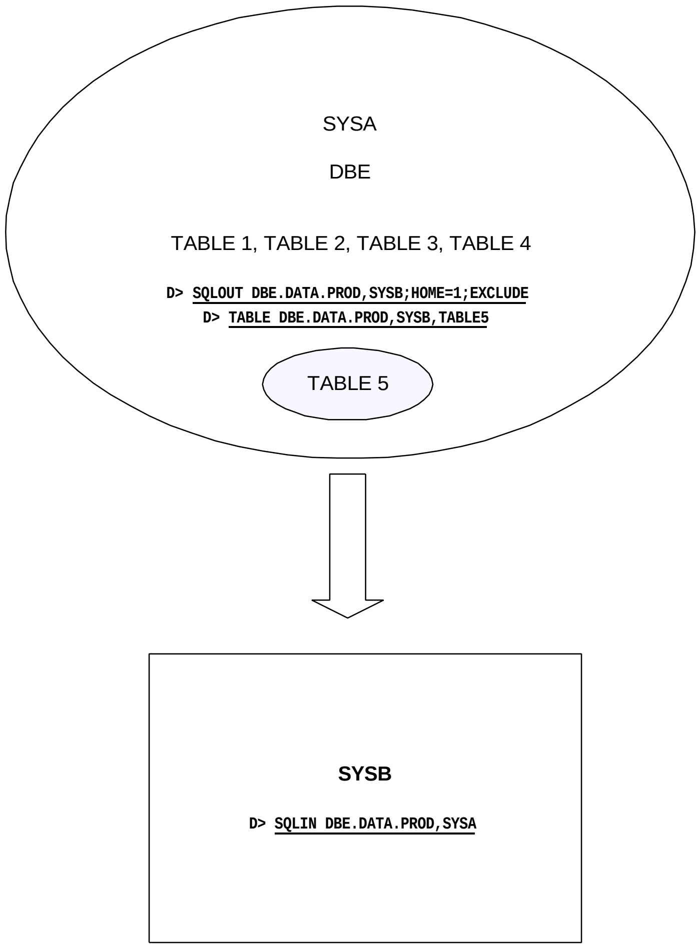
To exclude *table5* (shadowing the rest), you could enter:

On SYSA:

```
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=1;EXCLUDE
D> TABLE DBE.DATA.PROD,SYSB,TABLE5
```

On SYSB:

```
D> SQLIN DBE.DATA.PROD,SYSA
```


SYSA
DBE
TABLE 1, TABLE 2, TABLE 3, TABLE 4
D> SQLOUT DBE.DATA.PROD,SYSB;HOME=1;EXCLUDE
D> TABLE DBE.DATA.PROD,SYSB,TABLE5
TABLE 5
SYSB
D> SQLIN DBE.DATA.PROD,SYSA

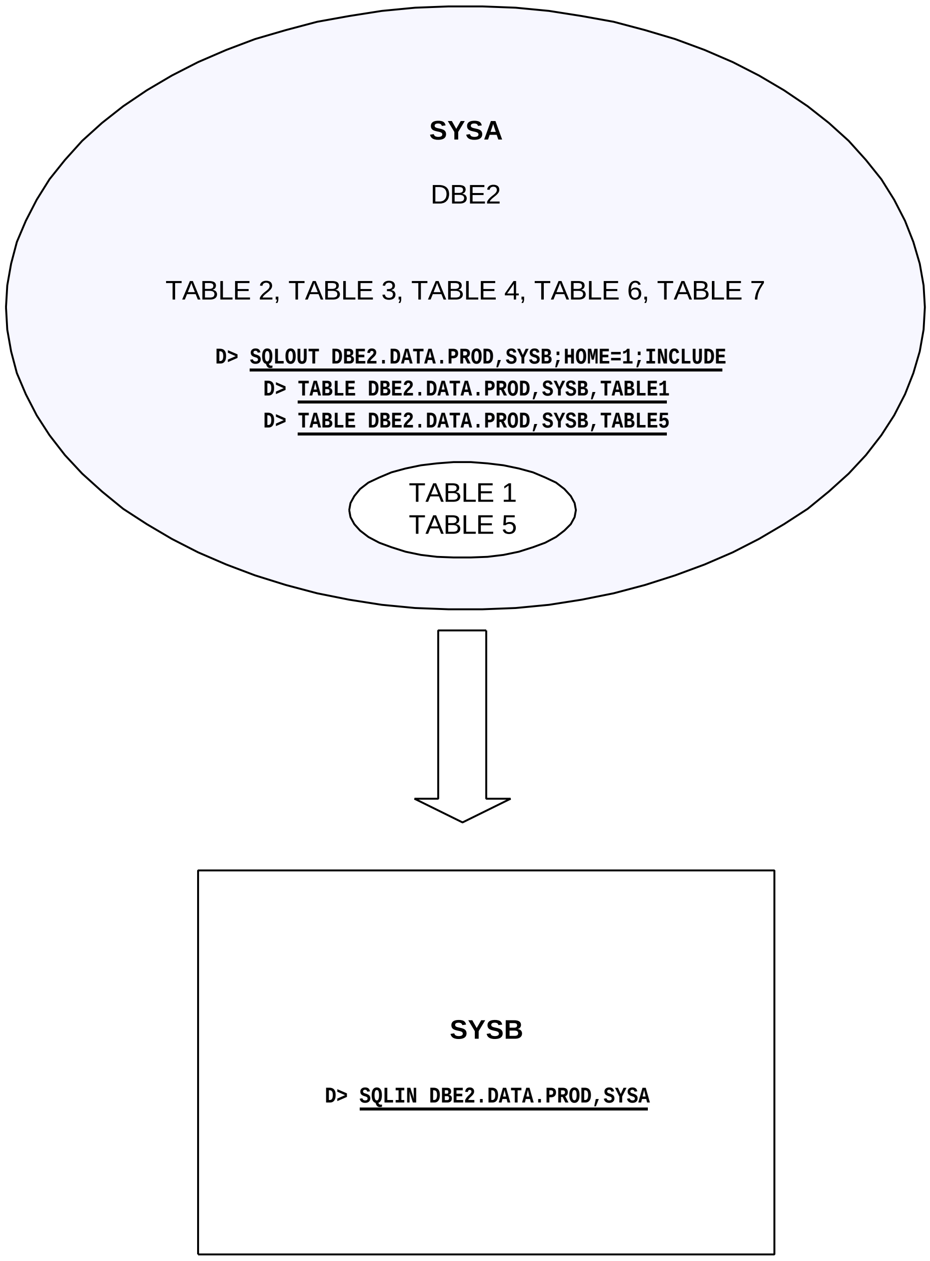
If DBE2 includes 7 tables, and you want to shadow only TABLE1 and TABLE 5, the INCLUDE option is your best choice:

On SYSA:

```
D> SQLOUT DBE2.DATA.PROD,SYSB;HOME=1;INCLUDE
D> TABLE DBE2.DATA.PROD,SYSB,TABLE1
D> TABLE DBE2.DATA.PROD,SYSB,TABLE5
```

On SYSB:

```
:NBDIR.PUB.NETBASE
D> SQLIN DBE2.DATA.PROD,SYSA
```


SYSA
DBE2
TABLE 2, TABLE 3, TABLE 4, TABLE 6, TABLE 7
D> SQLOUT DBE2.DATA.PROD,SYSB;HOME=1;INCLUDE
D> TABLE DBE2.DATA.PROD,SYSB,TABLE1
D> TABLE DBE2.DATA.PROD,SYSB,TABLE5
TABLE 1
TABLE 5
SYSB
D> SQLIN DBE2.DATA.PROD,SYSA

6. Enable Warm Standby Logging

Prior to enabling WSL, each DBE should be checked to verify that it has a valid WSL ID and Home Partition ID number, and the Maximum Number of Partitions is set. When an SQL database is created (via the START DBE NEW command in ISQL), by default these elements are initialized to zero. (Actually, WSL ID is initialized to blanks.) Each of these items must have bona fide values for standby logging to work.

1. To verify that these variables have valid values:

```
: SQLUTIL
>> SHOWDBE
DBEnvironment Name:  <<your dbe's name>>
Maintenance Word:        <CR>
Output File Name (opt):  <CR>
-> ALL
Maintenance word:
DBEnvironment Language:  NATIVE-3000
AutoStart:  ON
Warm Standby Logging Is: OFF
Warm Standby Logging ID is: MYDBE
Home partition ID Is: 1
Maximum number of Partitions Is: 2
User Mode:  MULTI
DBEFile0 Name:  CCDBES
DDL Enabled:  YES
No. of Runtime Control Block Pages:  37
No. of Data Buffer Pages:  240
Data Buffer Pages Memory Resident:  NO
No. of Log Buffer Pages:  120
Max. Transactions:  50
Maximum Timeout:  NONE
Default Timeout:  MAXIMUM
Authorize Once per Session:  OFF
-> E
>> E
```

If **WSL ID** is blank, if **Home partition ID** or **Maximum number of Partitions** is zero, you must modify the DBE.

Below are some tips on setting these values:

WSL ID	This name must be unique across the network. Once this is set, it should never be changed, unless a master DBE is being restored on a shadow machine. (Maximum 8 characters.)
Home partition ID	This must be a number between 1 and 32767 (inclusive), and each copy of the DBE must have a unique home partition number. Once this is set, it should never be changed.
Maximum number of Partitions	The number of partition instances with which the DBE may resync. This must be a number between 1 and 880 (inclusive). This number is used to calculate the size of the log file used by Standby Checkpoint Records. [MaxNumParts * 72 = number of bytes in the log file].

NOTE: Prior to enabling standby logging, the initial state of the DBE should be set up. All tables and authorization should be defined, and the database tables should be loaded.

To modify these values and enable Warm Standby Logging:

Enter the following:	**Syntax Notes:**
`: ISQL`	
`isql=> START DBE 'dbenvironmentName' NEWLOG <CR>`	No comma after NEWLOG
`> HOMEPARTITION = partitionID, <CR>`	
`> MAXPARTITIONS = maxnumparts, <CR>`	
`> WSLID = 'WSLID name', <CR>`	
`> STANDBY LOG <CR>`	No comma after STANDBY LOG
`> LOG DBEFILE oldtempnameforlog <CR>`	No comma after the old log file
`> WITH PAGES = logsizeinpages, <CR>`	name
`> NAME = 'newnameforlog'; <CR>`	
`isql=> GRANT DBA TO MGR@NETBASE; <CR>`	Semicolon at the end
`isql=> COMMIT WORK; <CR>`	Semicolon at the end
`isql=> BEGIN ARCHIVE; <CR>`	Semicolon at the end
`isql=> COMMIT ARCHIVE; <CR>`	Semicolon at the end
	Semicolon at the end

For example:	**Explanation:**
`: ISQL`	
`isql=> START DBE 'ourdbe.data' NEWLOG <CR>`	For the DBE named "ourdbe.data" (in the login account)
`> HOMEPARTITION = 2, <CR>`	use home partition ID "2",
`> MAXPARTITIONS = 3, <CR>`	maximum number of partitions of "3",
`> WSLID = 'SQLACE', <CR>`	and WSL ID "SQLACE".
`> STANDBY LOG <CR>`	Enable standby logging.
`> LOG DBEFILE sqltemp <CR>`	Use the temporary log file named "sqltemp,"
`> WITH PAGES = 5000, <CR>`	setting the log size to 5000 pages,
`> NAME = 'sqlalog.log'; <CR>`	and save the log file as a permanent log named "sqlalog.log."
`isql=> GRANT DBA TO MGR@NETBASE; <CR>`	Since the DBE was created by a user other than MGR.NETBASE, give MGR.NETBASE full access to the DBE.
`isql=> COMMIT WORK; <CR>`	Act on the previously entered commands.

`isql=> BEGIN ARCHIVE; <CR>`
`isql=> COMMIT ARCHIVE; <CR>`
	Invoke archive logging. (Must be perfomed on the master machine; optional on shadow systems.)

7. Add Additional Required Log Files

Once WSL is enabled, you have created a single log file. WSL requires a minimum of two log files. To add additional log files, you should run **SQLUtil**.

1. Run **SQLUtil**.

```
:SQLUTIL
>> ADDLOG
DBEnvironment Name:  <<your dbe's name>>
Maintenance Word:        <CR>
Name of Log file:     <<your log file's name>>
Size of Log file:     <<size in pages>>
Okay to add?           Y
>> EXIT
```

The ADDLOG command creates additional log files either in archive mode or non-archive mode, depending on the mode of the DBE. If your DBE is in archive mode (as your master DBE should be), the ADDLOG command will create archive log files. If your DBE is in non-archive mode (as your shadow machine may be), the resulting log files will be created as non-archive ones.

Non-archive log files are circular files which cannot be used for data recovery. Their size must be sufficient to handle the maximum number of concurrent transactions. You may use non-archive log files on your shadow systems only.

Archive log files are required for disaster recovery, and they are recommended on the master machine. Archive log files contain records for every addition, modification, or deletion of a tuple in the DBE. Archive log files retain this information until they are backed up using the STORE command in **SQLUtil**. Consequently, your system must have enough log files to accommodate all transactions between backups.

Sizing Log Files

If the space allocated for log files is inadequate, a **LOG FULL** condition will occur. If it is on the shadow side, the transactions will be queued on the master, but they will not be applied to the shadow. Additionally the import process on the shadow machine will be stopped. If LOG FULL happens on the master machine, processing will halt.

Sizes of archive and non-archive log files are stated in **log pages**. A log page is 512 bytes. You may build a maximum of 36 log files per DBE, and the maximum size of a log file is currently 524.287 pages.

The size of a non-archive log file may be calculated as follows:

[(Max. size of a transaction) x (Max. Number of Concurrent Transactions)] + 38 pages

Max. size of a transaction =
[(DBE size in pages) x (%inserted + %deleted + %changed) x 2] / 500 bytes per page

To calculate the size of archive log files, use the following formula:

[(Max size of a transaction) x (Max Transactions <u>per period</u>)] + 38 pages = Archive Logsize

You may build several log files to handle your data storage requirements (between backups), rather than two large ones. (Remember: Replicate requires a minimum of two log files.)

For more information on sizing log files, please consult your AllBase/SQL Database Administration Guide from Hewlett Packard.

To monitor the availability of log files (whether they are waiting for backup or are available for use), use the SHOWLOG command in **SQLUtil**.

8. Start SQL Shadowing

Once WSL is enabled, you may start and stop it within the NetBase control utility, **NBCTRL**.

If NetBase is running, to start SQL shadowing import and export processes for all configured nodes:

```
: NBCTRL.PUB.NETBASE
*> START SQL
*> EXIT
```

If NetBase is not running, you may start it which, in turn, will start SQL shadowing.

```
: NBCTRL.PUB.NETBASE
*> START NETBASE
*> EXIT
```

To stop SQL shadowing:

```
: NBCTRL.PUB.NETBASE
*> STOP SQL
*> EXIT
```

Below are examples of the few other commands you may enter within **NBCTRL** for more information on SQL shadowing.

If you wanted to start importing SQL transactions from a specific node, you could enter:

```
*> START SQLIN <node>
```

Or, if you wanted to start exporting SQL transactions to a specific node, you would enter:

```
*> START SQLOUT <node>
```

The same options are available on the **STOP** command. You may stop importing transactions from a specific node by entering:

```
*> STOP SQLIN <node>
```

Or, if you wanted to stop exporting SQL transactions to a specific node, you would enter:

```
*> STOP SQLOUT <node>
```

To review SQL shadowing:

```
*> SHOW SQL

Exporting Status:

  #  Node      DBE Name                  State   PIN

  7  CLIENT1   BRN1DBE.PUB.QDBEBRN1      ACTIVE   73
  8  DBSERVER  CCDBE.ROBIN.NETBASE       ACTIVE   39
  8  DBSERVER  XXDBE.ROBIN.NETBASE       INACT    47

Importing Status:

  #  Node      DBE Name                  State   PIN

  7  CLIENT1   BRN1DBE.PUB.QDBEBRN1      ACTIVE  119
  7  CLIENT1   FOO.PUB.NETBASE           ACTIVE  133
  7  CLIENT1   HRMSDB.HRMSDTA.REP01      INACT    99
  8  DBSERVER  CCDBE.SHADOW.NETBASE      DOWN
  8  DBSERVER  SHDBE.SHADOW.NETBASE      DOWN
```

In the above display the **STATE** field indicates the operating mode for the different exporting and importing nodes.

- **ACTIVE** indicates that the local process is running, as is the necessary remote process.
- **INACTIVE** identifies the situation in which the local process is running, but the remote process is not.
- **DOWN** indicates that the local process is not running.

Backups With WSL

When performing backups of an ALLBASE DBE, the SQL shadowing processes should be stopped. Both the export and import processes open the DBE and the log file.

The **STORE** command in **SQLUTIL** does not store the WSL log files; to accomplish that, the **STORELOG** command should be used. The **STORELOG** command stores a single (available) log file to tape.

WARNING! Do not use **STOREONLINE** to store the DBE while users are updating the DBE, because the stored log files are likely to be out of synchronization with the stored DBE.

Sample Startup and Backup Jobs

Sample jobs for starting SQL and backing it up may be found in the JOB group of the NetBase account. Please note that the jobs reflect a simple shadowing scenario from master to shadow, building archive log files on the master, and non-archive (small) log files on the shadow.

The naming convention is as follows:

SQLINITM.JOB.NETBASE	Start (initialize) SQL for the Master machine
SQLINITS.JOB.NETBASE	Start SQL for a shadow machine
SQLBACKM.JOB.NETBASE	Backup routine for the master machine
SQLBACKS.JOB.NETBASE	Backup routine for the shadow machine

These files are templates. They must be customized to work on your system, using your DBE names, home partition numbers, WSLIDs, account names, log file names, and sizes. The syntax for many of the commands is critical. Please modify the values, but maintain the same syntax, being careful to keep single quotes, commas, and semicolons where they are.

Chapter 3
User Exits

Overview

Four different types of user exits may be used with SQL shadowing; two are considered export user exits, and two are post user exits. If the USEREXIT option is specified on the **SQLOUT** command in **NBDIR**, the export process, **NBSQLEXP.NB**, will call the userexit (**export_sql_exit**) before sending the transaction to the import process on the other machine.

```
: NBDIR
D> SQLOUT dbe,node;PART=part1;...;USEREXIT
```

NetBase SQL shadowing has a way of allowing the customer to look at all updates to a DBE as they are made. The export process can be run so that it will read the transactions out of the logfile without sending them across to the shadow machine. In this case we say that the process is shadowing to a pseudo-node. In order to get all updates to the local DBE, the pseudo-node process must have a list of partition numbers; the home partition number and all the partition numbers for the nodes that transport their updates to this node.

If the node specified is "@", a pseudo node design is anticipated, and the USEREXIT option is assumed in the **SQLOUT** command. The export user exit, **exprt_sql_exit** is called, and if it contains logic for node zero (the pseudo-node), that logic is performed for the DBE.

```
D> SQLOUT dbe,@;PART=part1;...
```

If the USEREXIT option is specified on the **SQLIN** command in **NBDIR**, the import process, **NBSQLIMP.NB**, will call the user exit (**post_sql_exit_before**) before applying the transaction. Another post user exit (**post_sql_exit_after**) will be called if the apply fails.

```
D> SQLIN dbe,node;PART=part1;...;USEREXIT
```

User exits	Description
export_sql_exit	Export user exit for processing prior to sending the transaction to remote nodes. This exit may include logic for pseudo-nodes if it handles node zero (0).
post_sql_exit_before	Post user exit which processes the received transaction (on the "remote" node) prior to applying it to the DBE.
post_sql_exit_after	Post user exit which is invoked (on the "remote" node) if the apply fails. This user exit may send warnings to the console and retry the apply, or it may log the troublesome transaction and continue processing.

The names of the user exits must be as stated above, and they should reside in one of the following libraries:

XL.NB.NETBASE	XL.PUB.SYS	(not recommended)
XL.PUB.NETBASE		

The default library can be overridden by setting variables in NETBASE.JOB.NETBASE. You may override the location for **export_sql_exit** by setting EXPORTSQLPROC to the name of the library where the exit procedure resides. For **post_sql_exit_before** and **post_sql_exit_after** you may define a new library using the POSTSQLPROC variable. Examples of these statements are included in the job listed below.

```
!JOB NETBASE,MGR.NETBASE,PUB;HIPRI;OUTCLASS=,1;PRI=CS
!SETVAR EXPORTSQLPROC=XL.SPECIAL.PLACE
!SETVAR POSTSQLPROC=XL.SOMEWHER.ELSE
!SETJCW USEREXITTRACE=1
!
!COMMENT *** NETBASE job - Version 0.9.4 Upd 0
!
!CONTINUE
!PURGEGROUP TEMP
!CONTINUE
!NEWGROUP TEMP
!CONTINUE
!
!PURGE NBLOGBK.DATA
!CONTINUE
!RUN COPYN.LIB;INFO="FROM=NBLOG.DATA;TO=NBLOGBK.DATA"
!IF JCW >= FATAL THEN
!  CONTINUE
!  PURGE NBLOG.DATA
!  CONTINUE
!  BUILD NBLOG.DATA;REC=254,,V;DISC=256,1,1;CIR
!ENDIF
!SETDUMP
!
!CONTINUE
!RUN NBCOP.NB
!
!CONTINUE
!PURGEGROUP TEMP
!CONTINUE
!RUN PURGE.LIB
NBSE?#@.IPC
!EOJ
```

The user exits will be dynamically loaded by the import and export programs as needed. Consequently, in order to update the user exits, you must stop exporting and posting in order to unload the procedures and replace them.

All records that pass through the shadow export transport are sent to the **export_sql_exit** procedure. The user exit can determine the disposition of each record, and it can individually control logging if necessary. Since only one user exit procedure name is currently supported, the procedure must have the logic necessary to process all tables simultaneously. Typically, this requires simple switching logic to other procedures.

WSL Flags

Three flags in the WSL routines may be set by one of the NetBase processes or by a user exit. Normally, WSL performs several tests to validate an update before the update is applied on the shadow copy of the DBE. One check is to verify that the actual transaction length matches the transaction length in the transaction. If the user exit tells NetBase to delete one of the records in the transaction, NetBase will set the **NO_TRX_LEN_CHK** flag.

The WSL routines use before image information from the logfile to find the record being updated or deleted. In an environment performing concurrent updating of the same tuples, the user may want NetBase to set the **KEY_COL_SRCH_OK** flag. This flag tells WSL to search for the row using only the unique columns and the columns that have changed. The user can tell NetBase to set this flag through the USEREXIT.

The **post_sql_exit_after** procedure affects the NetBase's reaction when it encounters an error applying the transaction on the shadow copy. Normally, the posting process (**NBSQLIMP**) prints an error and terminates. If the USEREXIT flag is set on the **SQLIN** record and if a **post_sql_exit_after** procedure is found by the posting process, NetBase calls the user exit and continues applying the transaction. The default action is to delete the problem record and go on. If invoked, the user exit must decide what action should be taken and alert any other process if necessary.

The third flag that pertains to the WSL routines is set by NetBase if the **post_sql_exit_after** procedure is being used. It tells the apply procedure to, in the case of an error, return pointers to the record that caused the error. As a result, when NetBase calls the USEREXIT, it passes information about the problem record.

Notes on User Exit Structure

All of the user exits (both export user exits and post user exits) have the same set of parameters.

The ***sqlca*** parameter is the same as the ***sqlca*** as defined in the SQL manual for the language of your choice.

The ***comarea*** parameter has flags that are returned by the user exit. These flags control the processing of that record. The ***action*** field of the ***comarea*** should always be set by the user exit, since it is not initialized by the calling program. The ***comarea*** contains a large area (512 bytes) set aside for the user exit which is not in any way affected by the program calling it.

The ***header*** record contains all of the information about the record, the name of the DBE, the name of the Table (for UPDATE, INSERT, and DELETE records), the name of the user, and the time the record was originally committed. The ***header*** record also has pointers into the data records.

The ***data*** parameter has the record that was UDPATED, INSERTED, or DELETED.

The ***upd_data*** is used only for UPDATE records, and it contains the data which was updated.

A transaction consists of a BEGIN record, one or more INSERT, UPDATE, or DELETE records, and a COMMIT record. The user exit is called for each of these records. One of the options of the user exit is to delete the current record. It is not valid to delete a BEGIN or a COMMIT record.

Export Exit Procedure Calling Conventions

```
                 REC   REC     REC    REC  REC
export_sql_exit(sqlca,comarea,header,data,upd_data)
```

sqlca	HP SQL communication area for the DBE. This field can be used by the user exit to perform dynamic SQL commands on the DBE. See the SQL manual for the language in which the user exit is written for the definition of this field.
comarea	An array of various information used to control the exporting of shadow information. Some fields are reserved and must not be modified. Format is (C format):

```
struct  {
  int    action;
  int    status;
  int    error1;
  int    error2;
  int    export_node;
  int    import_node;
  union  {
     byte    all[16];
     struct  {
        byte    first_time;
        byte    key_col_search;
     } id;
  } flags;
  int      user_area[512];
} comarea;
```

The individual fields are defined as follows:

action Set by user exit procedure to cause various actions by the exporting process. The valid values are:

0	Process record normally. Record wil be sent to the specified destinations.
1	Ignore record completely. The record will be thrown away. This can easily cause synchronization errors.
2	Disable all subsequent calls to the user exit facility.
4	Exclude the table. Do not send updates for this table to the remote machine.
-1	Abort exporting. This is a drastic situation which can cause later sync errors.

status	Set by user exit procedure to send a message to the console. If a positive value is set, this message in the catalog file NBEXPSQL.CAT, set 5, will be sent to the console.
error1	Set by user exit procedure. If status is non-zero, these values will be ***error2*** inserted in the message sent to the console.
export_node	Set by calling procedure to indicate the node number from which transactions are being exported. This is the local node.
import_node	Set by calling procedure to indicate the node number to which transactions are being sent. This is the remote node. If the node is zero, it is assumed to be the pseudo-node for capturing transactions from a DBE without transferring them elsewhere.
first_time	This flag is set by the calling process. It is TRUE only the first time that the user exit is called.
key_col_search	This flag is set by the user exit. It tells the export process to set the **KEY_COL_SRCH_OK** flag when the transaction is applied.
user_area	This is a ***comarea*** provided to the USEREXIT for its own use.

header Contains all non-data information about the original update. This field contains the following:

```
struct {
  char   DBEname[26];
  char   owner_name[20];
  char   table_name[20];
  char   user_name[20];
  char   commit_time[24];
  short  record_type;
  short  num_columns;
  short  upd_columns;
  struct {
    short column_type;
    short column_length;
  } column_array[256];
  struct {
    short column_num;
    short column_type;
    short column_length;
  } update_array[256];
}
```

The individual fields within header are defined as follows:

DBEname	Name of the DBE (fully-qualified).
owner_name	Owner of the table.
table_name	Name of the table.
user_name	User name of the entity that committed the transaction. Only valid if record_type = 1.
commit_time	Time the record was committed. Only valid if record_type = 1. This field is in the same format as the SQL DATETIME data type.
record_type	Type of record. Defined as:

1	COMMIT TRANSACTION record
15	BEGIN TRANSACTION record
24	INSERT tuple
25	DELETE tuple
26	UPDATE tuple

num_columns	Number of columns in "data" buffer.
upd_columns	Number of columns in "upd_data" buffer. This will be 0 unless "record_type" is 26.
column_array	Describes all columns in the "data" buffer. The first entry describes the first column, the second describes column #2, etc.
column_type	Defined as one of the following:

-1	Column contains a NULL value
0	Binary data
1	Character data
2	Integer data
3	Floating point data
4	Packed decimal data
6	NLS character data

column_length	Length of the column in bytes.
update_array	Description of the updated columns. Only the columns being updated appear in this array, unlike the "column_array" which contains all columns.
column_num	Identifies the column number in the "upd_data" array.

Note: Currently, tables with up to 256 columns are supported.

dat	A buffer containing all columns of the record concatinated together. The description of the columns is found in the "column_array". For UPDATE records, the original data is stored here.
upd_data	A buffer containing all updated columns. Unlike the "data" buffer, this field contains only the updated columns. Note that if a column was not updated, it will not appear in this buffer. This is why the column number is specified in the "update_array" field.

Note: If a transaction contains updates that will be sent to more than one computer, it is necessary to send the BEGIN and COMMIT records to all remote nodes. Pairs of BEGIN and COMMIT transactions with no intervening updates will be ignored by the posting process.

Post Exit Procedure Calling Conventions

```
                      REC   REC     REC    REC  REC
post_sql_exit_before(sqlca,comarea,header,data,upd_data);

                      REC   REC     REC    REC  REC
post_sql_exit_after (sqlca,comarea,header,data,upd_data);
```

Parameters are defined as follows:

sqlca HP SQL communication area for the DBE. This field can be used by the user exit procedure to perform dynamic SQL commands on the DBE. See the HP SQL manual for the language in which the user exit is written for a description of this field.

comarea An array of various information used to control the posting of shadow information. Some fields are reserved and must not be modified. Format is (C format):

```
struct  {
  int    action;
  int    status;
  int    error1;
  int    error2;
  int    export_node;
  int    import_node;
  union  {
     byte    all[16];
     struct  {
        byte    first_time;
        byte    key_col_search;
     } id;
  } flags;
  int      user_area[512];
} comarea;
```

The individual fields are defined as follows:

action Set by user exit procedure to cause various actions by the exporting process. The valid values are:

0	Process record normally. Post the record as normal (before only) or continue with next record (after).
1	Ignore record completely. The record will be thrown away. This can easily cause synchronization errors. Used by "before" only.
2	Disable all subsequent calls to the user exit facility.
3	Roll back transaction. Ignore all subsequent updates in this transaction.
4	Do not post for this table. This is used if a table goes out of sync, but would allow posting of other tables.
5	Retry this transaction, setting UNIQUE CONSTRAINTS to "DEFERRED." ("after" only.)
6	Retry this transaction, setting REFERENCIAL CONSTRAINTS to "DEFERRED." ("after" only.)
7	Tells the import process to rollback and reapply. Used when the apply log fails because the pre-image could not be found and the post_sql_exit_after userexit fixes up the data in the buffer so that it will succeed. ("after" only.) **Note:** The import process cannot perform this function if the BEGIN record for the transaction is not in its internal buffers. In this case it will call the userexit again with the header.record_type set to 10000.
-1	Stop posting. This should normally be done only on a BEGIN or COMMIT transaction to prevent rollbacks.
-2	Do not post for this table. This is used if a table goes out of sync, but would allow posting of other tables. Sets the SYNC flag for the table.

status Set by user exit procedure to send a message to the console. If a positive value is set, this message in the catalog file NBEXPSQL.CAT, set 5, will be sent to the console.

error1 / error2 Set by user exit procedure. If status is non-zero, these values will be inserted in the message sent to the console.

export_node Set by calling procedure to indicate the node number that transactions are being exported from. This is the remote node.

import_node Set by calling procedure to indicate the node number that transactions are being applied to. This is the local node.

first_time	This flag is set by the calling process. It is TRUE only the first time that the user exit is called.
key_col_search	This flag is set by the user exit. It tells the import process to set the KEY_COL_SRCH_OK flag when the transaction is applied.
user_area	This is a comarea provided to the USEREXIT for its own use.
header	Contains all non-data information about the original update. This field contains the following:

```
struct {
  char   DBEname[26];
  char   owner_name[20];
  char   table_name[20];
  char   user_name[20];
  char   commit_time[24];
  short  record_type;
  short  num_columns;
  short  upd_columns;
  struct {
    short column_type;
    short column_length;
  } column_array[256];
  struct {
    short column_num;
    short column_type;
    short column_length;
  } update_array[256];
}
```

The individual fields are defined as follows:

DBEname	Name of the DBE (fully qualified).
owner_name	Owner of the table.
table_name	Name of the table.
user_name	User name of the entity that committed the transaction. Only valid if record_type = 1.
commit_time	Time the transaction was committed. Only valid if record_type = 1. This field is in the same format as SQLs DATETIME data type.
record_type	Type of record. Defined as:

1	COMMIT TRANSACTION record
15	BEGIN TRANSACTION record
24	INSERT tuple
25	DELETE tuple
26	UPDATE tuple

num_columns	Number of columns in "data" buffer.
upd_columns	Number of columns in "upd_data" buffer. This will be 0 unless "record_type" is 26.
column_array	Describes all columns in the "data" buffer. The first entry describes the first column, the second describes column #2, etc.
column_type	Defined as one of the following:

-1	Column contains a NULL value
0	Binary data
1	Character data
2	Integer data
3	Floating point data
4	Packed decimal data
6	NLS character data

column_length	Length of the column in bytes.
update_array	Description of the updated columns. Only the columns being updated appear in this array, unlike the "column_array" which contains all columns.
column_num	Identifies the column number in the "upd_data" array. **Note**: Currently, tables with up to 256 columns are supported.
data	A buffer containing all columns of the record concatinated together. The description of the columns is found in the "column_array". For UPDATE records, the original data is stored here.
upd_data	A buffer containing all updated columns. Unlike the "data" buffer, this field contains only the updated columns. Note that if a column was not updated, it will not appear in this buffer. This is why column number is specified in the "update_array" field.

Note: Error values returned be the posting procedure for the "after" user exit are described in the HP SQL manuals. Additional errors may be defined and will be added to this documentation.

Sample User Exit

The following sample user exit performs three functions:

1. It disables shadowing for the table named "LOCAL_TABLE".
2. Sets the WSL flag "KEY_COL_SEARCH_OK" on each update applied.
3. Prints a message to the console when the "CUST_NUM" column in the "CUSTOMERS" table is updated.

Your user exit could write a record to the DBE. If it does, that table should be excluded from shadowing.

```
#pragma list off

#define _MPEXL_SOURCE
#pragma intrinsic PRINT, PRINTOP
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpe.h>

#define byte unsigned char
#define MAX_COM_FLAGS 16
typedef struct {
      int      action;
      int      status;
      int      error1;
      int      error2;
      int      orignode;
      int      shadnode;
      union  {                  /* 16 bytes of flags */
          byte    all[MAX_COM_FLAGS];
          struct  {
             byte    first_time;
             byte    key_col_search;
          } id;
      } flags;
      int      user_area[512];
}  comarea_rec;

typedef struct {              /* column array record */
      short    column_type;
      short    column_length;
}   col_rec;

typedef struct {              /* update column record */
      short    upd_col_num;
      short    upd_col_type;
      short    upd_col_length;
}   upd_rec;
```

```
typedef struct {
      char     DBEname [26];
      char     owner_name [20];
      char     table_name [20];
      char     user_name [20];
      char     commit_time [24];
      short    record_type;
      short    num_columns;
      short    upd_columns;
      col_rec  column_array [256];
      upd_rec  update_array [256];
}  header_rec;

#pragma list on

void export_sql_exit (sqlca_type sqlca, comarea_rec *comarea,
                      header_rec *header, char *data, char *upd_data)
{
   int    rec,upd = 0;
   int    upd_offset = 0;
   int    rec_offset = 0;
   char   msg[130];

   comarea->action = 0;                           /* default to no action */

   if (comarea->shadnode == 0)  {

/*  THIS BLOCK HANDLES THE PSEUDO-NODE PROCESS                     */
/*  IT IS IMPORTANT TO KNOW WHEN THE COLUMN "CUST_NUM" IN THE      */
/*  "CUSTOMERS" TABLE IS MODIFIED.  PRINT TO THE CONSOLE IF IT IS. */
/*  CUST_NUM IS THE FOURTH COLUMN IN TABLE "CUSTOMERS".            */

      if (header->record_type == 26 &&               /* UPDATE */
          strncmp (header->table_name, "CUSTOMERS ", 10) == 0)  {
         while (upd < header->upd_columns &&
                header->update_array[upd].upd_col_num != 4)  {
            upd++;
            upd_offset += header->update_array[upd].upd_col_length;
         }
         if (upd < header->upd_columns)  {
            for (rec = 0;rec < 3; rec++)
               rec_offset += header->column_array[rec].column_length;
            strcpy (msg, "USEREXIT CUSTDBE : CUST_NUM.CUSTOMERS modified at ");
            memcpy (msg+strlen(msg), header->commit_time, 24);
            PRINTOP (msg, -strlen(msg), 0);
            strcpy (msg, "USEREXIT CUSTDBE : Pre-update ");
            memcpy (msg+30, data+rec_offset, 6);
            strcpy (msg+36, " Post-update: ");
            memcpy (msg+50, upd_data+upd_offset, 6);
            PRINTOP (msg, -56, 0);
         }
      }
      return;            /* This ends pseudo-node processing */
   }

/*  DISABLE SHADOWING FOR THE TABLE "LOCAL_TABLE"    */

   if (header->record_type != 15 &&               /* not BEGIN */
       header->record_type != 1 &&                /* not COMMIT */
       strncmp (header->table_name, "LOCAL_TABLE ", 12) == 0)
      comarea->action = 4;                        /* disable table */

/*  TABLE "GLOBAL_TABLE" IS SHADOWED BOTH WAYS AND IS PARTITIONED  */
/*  VERTICALLY.  ANY UPDATES TO THIS TABLE REQUIRE THAT THE        */
/*  KEY_COL_SEARCH_OK FLAG BE SET ON THE APPLY                     */

   if (header->record_type != 15 &&               /* not BEGIN */
       header->record_type != 1 &&                /* not COMMIT */
       strncmp (header->table_name, "GLOBAL_TABLE ", 13) == 0)
      comarea->flags.id.key_col_search = 1;       /* set KEY_COL_SEARCH_OK */

}  /* end export_sql_exit */

void post_sql_exit_before (sqlca_type sqlca, comarea_rec *comarea,
                           header_rec *header, char *data, char *upd_data)
{
   char  *loc_code;
   char   msg[130];

   comarea->action = 0;                           /* default to no action */

/*  TABLE "CUSTOMERS" IS SHADOWED BOTH WAYS AND IS PARTITIONED       */
/*  HORIZONTALLY.  RECORDS WITH A LOCATION-CODE OF "NW" MAY ONLY BE  */
/*  UPDATED HERE.  IF ANY OF THESE RECORDS COME IN FROM ANOTHER      */
/*  NODE, PRINT A MESSAGE TO THE CONSOLE AND THROW AWAY THE RECORD.  */
/*  LOCATION_CODE IS THE SECOND COLUMN IN CUSTOMERS.                 */

   if (header->record_type != 15 &&               /* not BEGIN */
       header->record_type != 1 &&                /* not COMMIT */
       strncmp (header->table_name, "CUSTOMERS ", 10) == 0)  {
      if (header->column_array[1].column_type == 1 &&      /* char data */
          header->column_array[1].column_length == 2)  {   /* length is 2 */
         loc_code = data + header->column_array[0].column_length;
         if (strncmp (loc_code, "NW", 2) == 0)  {
            sprintf (msg, "%s %d",
                     "USEREXIT CUSTDBE : LOCAL CUSTOMER MODIFIED BY NODE",
                     comarea->orignode);
            PRINTOP (msg, -strlen(msg), 0);
            comarea->action = 1;                  /* Throw away this record */
         }
      }
   }
}  /* end post_sql_exit_before */

void post_sql_exit_after  (sqlca_type sqlca, comarea_rec *comarea,
                           header_rec *header, char *data, char *upd_data)
{
```

```
EXEC SQL BEGIN DECLARE SECTION;
char        SQLMsg[130];
EXEC SQL END DECLARE SECTION;

   comarea->action = 0;

/*  IF IT'S NOT AN ERROR, RETURN                                   */

   if (sqlca.sqlcode >= 0)              /* Not an error */
      return;

/*  PRINT OUT ALL THE ERROR MESSAGES ASSOCIATED WITH THIS ERROR    */
/*  IF THE ERROR OCCURED ON AN INSERT, UPDATE OR DELETE THEN CHECK */
/*  THAT IT IS NOT AN INVALID TABLE                                */

   do  {                                /* Print the error */
      EXEC SQL SQLEXPLAIN :SQLMsg;
      PRINT (SQLMsg, -strlen(SQLMsg), 0);
      PRINTOP (SQLMsg, -strlen(SQLMsg), 0);
   } while (sqlca.sqlcode != 0);

   if (header->record_type >= 24 &&     /* INSERT, UPDATE, OR DELETE */
       header->record_type <= 26)  {
      if (header->record_type == 24)
         strcpy (SQLMsg, "INSERT ");
      else if (header->record_type == 25)
         strcpy (SQLMsg, "DELETE ");
      else if (header->record_type == 26)
         strcpy (SQLMsg, "UPDATE ");
      strcpy (SQLMsg+7, "error occured on table ");
      memcpy (SQLMsg+30, header->table_name, 20);
      SQLMsg[50] = '\0';
      PRINT (SQLMsg, -strlen(SQLMsg), 0);
      PRINTOP (SQLMsg, -strlen(SQLMsg), 0);

      if (strncmp (header->table_name, "LOCAL_TABLE ", 12) == 0)
         comarea->action = 4;           /* this table is disabled !!! */
   }
   else  {                              /* BEGIN OR COMMIT RECORD */
      if (header->record_type == 1)
         strcpy (SQLMsg, "Error occured on COMMIT record");
      else
         strcpy (SQLMsg, "Error occured on BEGIN record");
      PRINT (SQLMsg, -strlen(SQLMsg), 0);
      PRINTOP (SQLMsg, -strlen(SQLMsg), 0);
   }

}  /* end post_sql_exit_after  */
```

Sample User Exit

Below is a sample PASCAL user exit which performs the same functions as the previous example.

```
 INPUT-OUTPUT SECTION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 REC                 PIC S9(09) COMP VALUE 1.
 01 UPD                 PIC S9(09) COMP VALUE 1.
 01 UPD-OFFSET          PIC S9(09) COMP VALUE 1.
 01 REC-OFFSET          PIC S9(09) COMP VALUE 1.
 01 MESSAGE-BUFFER      PIC X(130).
 01 HOLD-AREA           PIC X(6).
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 SQLMESSAGE          PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
 LINKAGE SECTION.
 EXEC SQL INCLUDE SQLCA END-EXEC.
 01 SQLCA-ARRAY.
    05 SQLCA-ENTRY      PIC S9(09) COMP OCCURS 500 TIMES.
 01 COMAREA.
    05 ACTION           PIC S9(09) COMP.
    05 COMAREA-STATUS   PIC S9(09) COMP.
    05 ERROR1           PIC S9(09) COMP.
    05 ERROR2           PIC S9(09) COMP.
    05 ORIGNODE         PIC S9(09) COMP.
    05 SHADNODE         PIC S9(09) COMP.
    05 FLAGS            PIC X(16).
    05 FLAGS-REDF REDEFINES FLAGS.
       10 FIRST-TIME     PIC X(01).
       10 KEY-COL-SEARCH PIC X(01).
       10 FILLER         PIC X(14).
    05 USER-AREA        PIC S9(09) COMP OCCURS 512 TIMES.
 01 HEADER.
    05 DBENAME          PIC X(26).
    05 OWNER-NAME       PIC X(20).
    05 TABLE-NAME       PIC X(20).
    05 USER-NAME        PIC X(20).
    05 COMMIT-TIME      PIC X(24).
    05 RECORD-TYPE      PIC S9(04) COMP.
    05 NUMBER-COLUMNS   PIC S9(04) COMP.
    05 UPDATE-COLUMNS   PIC S9(04) COMP.
    05 HDR-COLUMN-ARRAY     OCCURS 256 TIMES.
       10 COLUMN-TYPE    PIC S9(04) COMP.
       10 COLUMN-LENGTH  PIC S9(04) COMP.
    05 HDR-UPDATE-ARRAY     OCCURS 256 TIMES.
       10 UPD-COL-NUM    PIC S9(04) COMP.
       10 UPD-COL-TYPE   PIC S9(04) COMP.
       10 UPD-COL-LENGTH PIC S9(04) COMP.
 01 UPDATE-ARRAY        PIC X(2000).
 01 DATA-ARRAY          PIC X(2000).
$PAGE
```

```
*******************************************************************
*                                                                 *
*                        PROCEDURE DIVISION                       *
*                                                                 *
*******************************************************************
 PROCEDURE DIVISION.

 000-MAIN-LOGIC SECTION 01.
 000-EXIT.
     EXIT.
$PAGE
 A000-EXPORT-SQL-EXIT SECTION 01.

     ENTRY "export_sql_exit" USING SQLCA-ARRAY,
                                   COMAREA,
                                   HEADER,
                                   DATA-ARRAY,
                                   UPDATE-ARRAY.
     MOVE ZERO                   TO ACTION
                                 IN COMAREA.
     IF SHADNODE IN COMAREA = 0 THEN
* Display when customers dataset is modified. CUST_NUM is the
* forth column in table customers.
        IF RECORD-TYPE IN HEADER = 26 AND
           TABLE-NAME  IN HEADER = "CUSTOMERS " THEN
           MOVE ZERO             TO UPD-OFFSET
           PERFORM A100-COUNT-UPDATES
             UNTIL UPD > UPDATE-COLUMNS IN HEADER AND
                   UPD-COL-NUM IN HDR-UPDATE-ARRAY(UPD) NOT = 4
           IF UPD < UPDATE-COLUMNS IN HEADER THEN
              PERFORM A200-INCREMENT-REC-COUNT
                       UNTIL REC = 4
              PERFORM A300-DISPLAY-DATA
              GOBACK
           ELSE
              PERFORM A300-DISPLAY-DATA
              GOBACK
        ELSE
            GOBACK.
* Disable shadowing for the table "LOCAL_TABLE"
     IF RECORD-TYPE IN HEADER NOT = 15 AND
        RECORD-TYPE IN HEADER NOT = 1  AND
        TABLE-NAME  IN HEADER     = "LOCAL_TABLE " THEN
        MOVE 4                   TO ACTION
                                 IN COMAREA.
* Table GLOBAL_TABLE is shadowed both ways and is partitioned
* vertically. Any updates to this table require that the
* KEY_COL_SEARCH_OK flag be set on the apply.
     IF RECORD-TYPE IN HEADER NOT = 15 AND
        RECORD-TYPE IN HEADER NOT = 1  AND
        TABLE-NAME  IN HEADER     = "GLOBAL_TABLE " THEN
        MOVE 1                   TO KEY-COL-SEARCH
                                 IN FLAGS-REDF.
     GOBACK.
 A00A-EXIT.
      EXIT.
$PAGE
 A100-COUNT-UPDATES SECTION 01.
     COMPUTE UPD-OFFSET = UPD-OFFSET +
             UPD-COL-LENGTH IN HDR-UPDATE-ARRAY(UPD).
     ADD 1                       TO UPD.
 A100-EXIT.
      EXIT.
$PAGE
 A200-INCREMENT-REC-COUNT SECTION 01.
     COMPUTE REC-OFFSET = REC-OFFSET +
             COLUMN-LENGTH IN HDR-COLUMN-ARRAY(REC).
     ADD 1                       TO REC.
 A200-EXIT.
      EXIT.
$PAGE
 A300-DISPLAY-DATA SECTION 01.

     DISPLAY "USEREXIT CUSTDBE : CUST_NUM.CUSTOMERS modified at ",
             COMMIT-TIME IN HEADER.

     MOVE DATA-ARRAY(REC-OFFSET:6)
                                 TO HOLD-AREA.
     DISPLAY "USEREXIT CUSTDBE : Pre-update ",HOLD-AREA.

     MOVE UPDATE-ARRAY(REC-OFFSET:)
                                 TO HOLD-AREA.
     DISPLAY "USEREXIT CUSTDBE : Post-update ",HOLD-AREA.

 A300-EXIT.
      EXIT.
$PAGE
 B000-POST-SQL-EXIT-BEFORE SECTION 01.

     ENTRY "post_sql_exit_before" USING SQLCA-ARRAY,
                                        COMAREA,
                                        HEADER,
                                        DATA-ARRAY,
                                        UPDATE-ARRAY.
     MOVE ZERO                   TO ACTION
                                 IN COMAREA.
* Table "CUSTOMERS" is shadowed both ways and is partitioned
* horizontally. Records with a LOCATION-CODE of "NW" may only be
* updated here. If any of these records come in from another
* node, print a message and throw away the record. LOCATION-CODE
* is the second column in table CUSTOMERS.
     IF RECORD-TYPE IN HEADER NOT = 15 AND
        RECORD-TYPE IN HEADER NOT = 1  AND
        TABLE-NAME  IN HEADER     = "CUSTOMERS " THEN
        IF COLUMN-TYPE   IN HDR-COLUMN-ARRAY(2) = 1  AND
           COLUMN-LENGTH IN HDR-COLUMN-ARRAY(2) = 2  THEN
           COMPUTE REC-OFFSET =
                COLUMN-LENGTH IN HDR-COLUMN-ARRAY(1) + 1
           IF DATA-ARRAY(REC-OFFSET:2) = "NW" THEN
            DISPLAY "USEREXIT CUSTDBE:LOCAL CUST_NUM MODIFIED ",
                       ORIGNODE  IN COMAREA
              MOVE 1             TO ACTION IN COMAREA
           ELSE
              NEXT SENTENCE
        ELSE
           NEXT SENTENCE
     ELSE
        NEXT SENTENCE.

     GOBACK.
 B00A-EXIT.
      EXIT.
$PAGE
 C000-POST-SQL-EXIT-AFTER SECTION 01.

     ENTRY "post_sql_exit_after" USING SQLCA,
                                       COMAREA,
                                       HEADER,
                                       DATA-ARRAY,
                                       UPDATE-ARRAY.
     MOVE ZERO                   TO ACTION
                                 IN COMAREA.
* Return if not an error
     IF SQLCODE IN SQLCA = 0 THEN GOBACK.
* Display error message associated with error.
     EXEC SQL SQLEXPLAIN : SQLMESSAGE END-EXEC.
     DISPLAY SQLMESSAGE.
* Display type of transaction.
     IF RECORD-TYPE IN HEADER >= 24 AND
        RECORD-TYPE IN HEADER <= 26 THEN
        PERFORM C010-DISPLAY-TRANSACTION
        DISPLAY "Error occurred on table ",
                   TABLE-NAME  IN HEADER
        IF TABLE-NAME IN HEADER = "LOCAL_TABLE " THEN
           MOVE 4              TO ACTION
                               IN COMAREA
        ELSE
           NEXT SENTENCE
     ELSE
        IF RECORD-TYPE         IN HEADER = 1
           DISPLAY "Error occurred on COMMIT record"
        ELSE
           DISPLAY "Error occurred on BEGIN record".
     GOBACK.
 C00A-EXIT.
      EXIT.
$PAGE
 C010-DISPLAY-TRANSACTION SECTION 01.
     IF RECORD-TYPE IN HEADER = 24 THEN
        DISPLAY "INSERT"
     ELSE
     IF RECORD-TYPE IN HEADER = 25 THEN
```

```
              DISPLAY "DELETE"
           ELSE
           IF RECORD-TYPE IN HEADER = 26 THEN
              DISPLAY "UPDATE".
       C010-EXIT.
            EXIT.
```

A sample user exit for batch update shadowing which alerts the shadow system that "all" transactions have been processed by creating a "STOP" MPE file. In addition, the user exit explains with which transaction it struggled to update the shadow copy (if any).

```
#pragma list off

#define _MPEXL_SOURCE
#pragma intrinsic FOPEN, FCHECK, FCLOSE, PRINT
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpe.h>

#define TRUE  1
#define FALSE 0
#define CHR0  '\0'
#define CCE   2

#define BEGIN  1
#define COMMIT 15
#define INSERT 24
#define DELETE 25
#define UPDATE 26

#define DATE 10
#define TIME 11
#define DATETIME 12

#define byte unsigned char
#define MAX_COM_FLAGS 16
typedef struct {
      int      action;
      int      status;
      int      error1;
      int      error2;
      int      orignode;
      int      shadnode;
      union  {                   /* 16 bytes of flags */
          byte    all[MAX_COM_FLAGS];
          struct  {
             byte    first_time;
             byte    key_col_search;
          } id;
      } flags;
      int      user_area[512];
}  comarea_rec;

typedef struct {              /* column array record */
      short    column_type;
      short    column_length;
}   col_rec;

typedef struct {              /* update column record */
      short    upd_col_num;
      short    upd_col_type;
      short    upd_col_length;
}   upd_rec;

typedef struct {
      char     DBEname [26];
      char     owner_name [20];
      char     table_name [20];
      char     user_name [20];
      char     commit_time [24];
      short    record_type;
      short    num_columns;
      short    upd_columns;
      col_rec  column_array [256];
      upd_rec  update_array [256];
}  header_rec;

#pragma list on

EXEC SQL BEGIN DECLARE SECTION;
char        SQLMsg[130];                 /* Used by SQLExplain */
char        tableName[20];               /* Used by various SQL commands */
EXEC SQL END DECLARE SECTION;

/*                    UTILITY FUNCTIONS                        */

/* uexit_datetime_convert                                      */
/*                                                             */
/* Convert the DATE/TIME/DATETIME data type from the internal   */
/* format into ascii.  Return a pointer to the ascii string.    */
/*                                                             */
char *uexit_datetime_convert (byte *data, int type)
{
   static char result[40];
   struct  {
      unsigned  year :14;
      unsigned  month :4;
      unsigned  day :6;
   } dt;

   struct  {
      unsigned  hour :6;
      unsigned  minute :6;
      unsigned  second :6;
   } tm;

   struct  {
      unsigned  fill :2;
      unsigned  milli :20;
      unsigned  filler :2;
   } mi;

   switch (type)  {
      case DATE:
         memcpy (&dt, data, sizeof(dt));
         sprintf (result, "%4.4d-%2.2d-%2.2d",
                           dt.year, dt.month, dt.day);
         break;

      case TIME:
         memcpy (&tm, data+3, sizeof(tm));
         sprintf (result, "%2.2d:%2.2d:%2.2d",
                           tm.hour, tm.minute, tm.second);
         break;

      case DATETIME:
         memcpy (&dt, data, sizeof(dt));
         memcpy (&tm, data+3, sizeof(tm));
         memcpy (&mi, data+5, sizeof(mi));
         sprintf (result,
                  "%4.4d-%2.2d-%2.2d %2.2d:%2.2d:%2.2d.%3.3d",
                  dt.year, dt.month, dt.day, tm.hour,
                  tm.minute, tm.second, mi.milli);
         break;

      default:
         break;

   }
   return result;

}   /* end uexit_datetime_convert */


/* uexit_BCD_convert                                            */
/*                                                              */
/* Convert the passed data from BCD to ASCII.  The function     */
/* returns a pointer to the ASCII string.                       */
/*                                                              */
char *uexit_BCD_convert (char *data, short length, short precision,
                         short scale)
{
#define ASCIIZero   '0'
#define MinusSign   13
#define btod(d,i)   ((i&1)?((d[i/2])&0xf):((d[i/2]>>4)&0xf))

   int     i;
   int     DecimalPlace;
   int     PutPos = 0;
   int     DataEnd;
   int     DataStart;
   int     SignPos;
   static
   char    result[80];
   char    *outStr = result + 2;

   DataEnd = (length * 2) - 2;
   DataStart = DataEnd - precision + 1;
   SignPos = DataEnd + 1;
   memset (result, 0, 80);
   if ((DecimalPlace = precision - scale) == 0)
      outStr[PutPos++] = '.';

   for (i = DataStart; i <= DataEnd; i++)  {
      outStr[PutPos] = ASCIIZero + btod(data,i);
      if (PutPos == DecimalPlace-1)
         outStr[++PutPos] = '.';
      PutPos++;
   }

   for (;*outStr && *outStr == '0'; outStr++);
   if (*outStr == 0)
      *outStr = '0';
   else  {
      if (*outStr == '.')
         *(--outStr) = '0';
      if (btod(data,SignPos) == MinusSign)
         *(--outStr) = '-';
   }

   return outStr;

}  /* end uexit_BCD_convert */


/* uexit_print_data                                            */
/*                                                             */
/* Format and then print the data.  The type of data and its    */
/* length are passed in.                                       */
/*                                                             */
void uexit_print_data (char *data, char *cname, short ctype,
                       int clen, int precision, int scale)
{
   char       msg[90];
   short      sdata;
   int        idata;
   struct  {
      unsigned  year :14;
      unsigned  month :4;
      unsigned  day :6;
   } dt;

   struct  {
      unsigned  hour :6;
      unsigned  minute :6;
      unsigned  second :6;
   } tm;

   struct  {
      unsigned  fill :2;
      unsigned  milli :20;
      unsigned  filler :2;
   } mi;

   memcpy (msg, cname, 20);
   memcpy (msg+20, "  ", 2);
   switch (ctype)  {

      case 0:
         if (clen == 2)  {
            memcpy (&sdata, data, 2);
            sprintf (msg+22, "%d", sdata);
         }
         else
         if (clen == 4)  {
            memcpy (&idata, data, 4);
            sprintf (msg+22, "%d", idata);
         }
         else
            sprintf (msg+22, "CANNOT DISPLAY INTEGER DATA, LEN %d", clen);
         PRINT (msg, -strlen(msg), 0);
         break;

      case 2:
         memcpy (msg+22, data, clen);
         PRINT (msg, -(22+clen), 0);
         break;

      case 3:
         if (clen > 0)
            memcpy (msg+22, data, clen);
         PRINT (msg, -(22+clen), 0);
         break;

      case 5:
         strcpy (msg+22, uexit_BCD_convert(data,
                      clen, precision, scale));
         PRINT (msg, -strlen(msg), 0);
         break;

      case DATE:
         strcpy (msg+22, uexit_datetime_convert (data, DATE));
         PRINT (msg, -strlen(msg), 0);
         break;

      case TIME:
         strcpy (msg+22, uexit_datetime_convert (data, TIME));
         PRINT (msg, -strlen(msg), 0);
         break;

      case DATETIME:
         strcpy (msg+22, uexit_datetime_convert (data, DATETIME));
         PRINT (msg, -strlen(msg), 0);
         break;
      default:
         sprintf (msg+22, "CANNOT DISPLAY DATATYPE %d", ctype);
         PRINT (msg, -strlen(msg), 0);
         break;
   }

}   /* end uexit_print_data */


/* uexit_process_data                                          */
/*                                                             */
/* Get the column information for the column number passed and  */
/* print the data.  The function returns FALSE if processing    */
/* should stop.                                                */
/*                                                             */
int uexit_process_data (sqlca_type sqlca, char *data, int *offset,
                        int col, int lengthUsed, int *haveVAR)
{
EXEC SQL BEGIN DECLARE SECTION;
char        columnName[21];
int         colNumber;
int         clen;
short       ctype;
short       precision;
short       scale;
EXEC SQL END DECLARE SECTION;

   char       msg[90];

   colNumber = col;
   EXEC SQL SELECT COLNAME, LENGTH, TYPECODE, PRECISION, SCALE
            INTO   :columnName, :clen, :ctype, :precision, :scale
            FROM   SYSTEM.COLUMN
            WHERE  COLNUM = :colNumber AND TABLENAME = :tableName;
   if (sqlca.sqlcode == 100)  {
      sprintf(msg, "NO ROWS QUALIFIED - SELECT OF SYSTEM.COLUMN FOR %d",
              col);
      PRINT (msg, -(strlen(msg)), 0);
      return FALSE;
   }
   else
   if (sqlca.sqlcode < 0)  {
      sprintf (msg, "ERROR GETTING COLUMN INFORMATION FOR %d",col);
      PRINT (msg, -(strlen(msg)), 0);
      PRINT (msg, 0, 0);
      do  {                                /* Print the error */
         EXEC SQL SQLEXPLAIN :SQLMsg;
         PRINT (SQLMsg, -strlen(SQLMsg), 0);
      } while (sqlca.sqlcode != 0);
      return FALSE;
   }

   if (ctype == 3)  {                      /* VARCHAR */
      clen = lengthUsed;
```

```
         *haveVAR = TRUE;
      }
      else
      if (*haveVAR && lengthUsed == 0)  {
         strcpy (msg, "CANNOT DECODE DATA, NO COLUMN INFO AFTER VARCHAR");
         PRINT (msg, -(strlen(msg)), 0);
         return FALSE;
      }
      else
      if (lengthUsed > 0 &&
          lengthUsed != clen)  {
         strcpy (msg, "CANNOT DECODE DATA, COLUMN LENGTHS DO NOT MATCH");
         PRINT (msg, -(strlen(msg)), 0);
         return FALSE;
      }

      uexit_print_data (data+*offset, columnName, ctype,
                        clen, precision, scale);
      *offset += clen;
      return TRUE;

   }   /* end of uexit_process_data */


   /*                        USEREXITS                           */

   /* export_sql_exit                                             */
   /*                                                             */
   /* If the UPDATE or INSERT is for the STATUS table, get the     */
   /* record and validate.  If the record is valid, create a file  */
   /* using the SOURCE column of the status record in the name.    */
   /*                                                             */
   void export_sql_exit (sqlca_type sqlca, comarea_rec *comarea,
                         header_rec *header, char *data, char *upd_data)
   {
   EXEC SQL BEGIN DECLARE SECTION;
   struct {
         char     status_name [20];
         char     source [4];
         char     source_desc [16];
         char     business_date [16];
         char     start_time [16];
         char     finish_time [16];
   }  log_data;
      char     source_desc [17];
      char     business_date [11];
      char     start_time [24];
      char     finish_time [24];
      char     log_status_name [20];
      char     log_source [4];
   EXEC SQL END DECLARE SECTION;

      int    log_offset, upd_offset;
      int    unum, i, l;
      char   fname[40];
      short  err;
      char   msg[80];
      char  *logp = (char *)&log_data;

      comarea->action = 0;                   /* Default to no action */

   /*  PROCESS ONLY IF STATUS TABLE AND INSERT OR DELETE RECORD */

      if (memcmp (header->table_name, "STATUS  ", 8) != 0 ||
          (header->record_type != INSERT &&    /* Check the status record */
           header->record_type != UPDATE))
         return;

   /*  GET THE NEW STATUS RECORD */

      memcpy (logp, data, sizeof(log_data));
      if (header->record_type == UPDATE)  {   /* Move in the updated cols */
         l = i = upd_offset = log_offset = 0;
         for (; l < header->upd_columns; l++)  {
            unum = header->update_array[l].upd_col_num;
            for (; i < unum-1; i++)
               log_offset += header->column_array[i].column_length;
            memcpy (logp+log_offset, upd_data+upd_offset,
                    header->update_array[l].upd_col_length);
            upd_offset += header->update_array[l].upd_col_length;
         }
      }

   /*  READ THE STATUS RECORD FROM THE DBE  */

      memcpy (log_status_name, log_data.status_name, 20);
      memcpy (log_source, log_data.source, 4);
      EXEC SQL SELECT   SOURCE_DESC,
                        BUSINESS_DATE,
                        START_TIME,
                        FINISH_TIME
               INTO     :source_desc,
                        :business_date,
                        :start_time,
                        :finish_time
               FROM     STATUS
               WHERE    STATUS_NAME = :log_status_name
               AND      SOURCE      = :log_source;

      if (sqlca.sqlcode == 0 &&
          memcmp (source_desc, log_data.source_desc, 16) == 0 &&
          strcmp (business_date,
                  uexit_datetime_convert(log_data.business_date,DATE)) == 0 &&
          strcmp (start_time,
                  uexit_datetime_convert(log_data.start_time,DATETIME)) == 0 &&
          strcmp (finish_time,
                  uexit_datetime_convert(log_data.finish_time,DATETIME)) == 0) {
```

```
/*  CREATE THE MPE FILE STOP<SOURCE>.PUB.NETBASE  */

      strcpy (fname, "STOP");
      for (i = 0; i < 4 && log_source[i] != ' '; i++);
      memcpy (fname+4, log_source, i);
      fname[i+4] = CHR0;
      strcat (fname, ".PUB.NETBASE ");

      i = FOPEN (fname, 0);
      if (ccode() != CCE)  {               /* Open of the file failed */
         FCHECK (i, &err);
         sprintf (msg, "Error %d opening file %s", err, fname);
         PRINT (msg, -strlen(msg), 0);
      }
      else  {                              /* Open succeeded, close perm */
         FCLOSE (i, 1, 0);
         if (ccode() != CCE)  {            /* Close failed */
            FCHECK (i, &err);
            if (err != 100)  {             /* Duplicate permanent file */
               sprintf (msg, "Error %d creating file %s", err, fname);
               PRINT (msg, -strlen(msg), 0);
            }
            else  {
               sprintf (msg, "Cannot create %s, file already exists",
                        fname);
               PRINT (msg, -strlen(msg), 0);
            }
         }
      }
   }  /* end of if select record found */

}  /* end export_sql_exit */


/* post_sql_exit_qfter                                          */
/*                                                              */
/* Print out the type of record and call SQLEXPLAIN for all     */
/* errors.  If the record is an UPDATE, INSERT or DELETE,       */
/* format and print out hte data.                          .    */
/*                                                              */
void post_sql_exit_after (sqlca_type sqlca, comarea_rec *comarea,
                          header_rec *header, char *data, char *upd_data)
{
EXEC SQL BEGIN DECLARE SECTION;
int         numColumns;
EXEC SQL END DECLARE SECTION;

   int        l, i, mlen;
   int        col;
   char       msg[90];
   int        offset;
   int        haveVAR = FALSE;

   comarea->action = 0;
```

```
/*  IF IT'S NOT AN ERROR, RETURN                                   */

   if (sqlca.sqlcode >= 0)              /* Not an error */
      return;

/*  PRINT A HEADER MESSAGE */

   sprintf (SQLMsg, "POST_SQL_EXIT_AFTER (%d) ", header->record_type);
   if (header->record_type == COMMIT)
      strcat (SQLMsg, "COMMIT WORK ");
   else
   if (header->record_type == BEGIN)
      strcat (SQLMsg, "BEGIN WORK ");
   else
   if (header->record_type == INSERT)
      strcat (SQLMsg, "INSERT INTO ");
   else
   if (header->record_type == DELETE)
      strcat (SQLMsg, "DELETE FROM ");
   else
   if (header->record_type == UPDATE)
      strcat (SQLMsg, "UPDATE ");
   else
      strcat (SQLMsg, "UNKNOWN ");

   if (header->record_type >= 24 &&        /* INSERT, UPDATE or DELETE */
       header->record_type <= 26)  {
      for (l = 0; l < 20 && header->table_name[l] != ' '; l++);
      mlen = strlen(SQLMsg);
      memcpy (SQLMsg+mlen, header->table_name, l);
      mlen += l;
      SQLMsg[mlen++] = '/';
      for (l = 0; l < 20 && header->owner_name[l] != ' '; l++);
      memcpy (SQLMsg+mlen, header->owner_name, l);
      SQLMsg[mlen+l] = CHR0;
   }

   PRINT (SQLMsg, -strlen(SQLMsg), 0);

/*  LET SQLEXPLAIN DISPLAY ALL THE ERRORS  */

   do  {                                /* Print the error */
      EXEC SQL SQLEXPLAIN :SQLMsg;
      PRINT (SQLMsg, -strlen(SQLMsg), 0);
   } while (sqlca.sqlcode != 0);

/*  IF BEGIN OR COMMIT, OUR WORK IS DONE */

   if (header->record_type == BEGIN ||
       header->record_type == COMMIT ||
       header->num_columns == 0)
      return;
```

```
/*  READ THE NUMBER OF COLUMNS FOR THIS TABLE  */

   memcpy (tableName, header->table_name, 20);
   EXEC SQL SELECT NUMC
            INTO   :numColumns
            FROM   SYSTEM.TABLE
            WHERE  NAME = :tableName;

   if (sqlca.sqlcode == 100)  {
      strcpy (msg, "NO ROWS QUALIFIED - SELECT OF SYSTEM.TABLE ");
      memcpy (msg+(mlen=strlen(msg)), tableName, 20);
      PRINT (msg, -(mlen+20), 0);
      return;
   }
   else
   if (sqlca.sqlcode < 0)  {
      strcpy (msg, "ERROR GETTING NUMBER OF COLUMNS FOR ");
      memcpy (msg+(mlen=strlen(msg)), tableName, 20);
      PRINT (msg, -(mlen+20), 0);
      PRINT (msg, 0, 0);
      do  {                                /* Print the error */
         EXEC SQL SQLEXPLAIN :SQLMsg;
         PRINT (SQLMsg, -strlen(SQLMsg), 0);
      } while (sqlca.sqlcode != 0);
      return;
   }

/*  LOOP THROUGH THE COLUMN ARRAY, GETTING THE NAME AND OTHER INFO  */
/*  THEN PRINTING THE FORMATTED DATA                                */

   PRINT (msg, 0, 0);
   for (l = 0, offset = 0; l < header->num_columns; l++)  {

      if ((col = l + 1) > numColumns)  {
         sprintf (msg, "COLUMN %d NOT DEFINED", col);
         PRINT (msg, -strlen(msg), 0);
         continue;
      }

      uexit_process_data (sqlca, data, &offset, col,
                          header->column_array[l].column_length, &haveVAR);

   }  /* end of column array loop */
   PRINT (msg, 0, 0);

   if (header->record_type == UPDATE)  {
      strcpy (msg, "UPDATED TO:");
      PRINT (msg, -strlen(msg), 0);
      PRINT (msg, 0, 0);
      for (l = 0, offset = 0; l < header->upd_columns; l++)  {

         if (header->update_array[l].upd_col_num == 0 &&
             header->update_array[l].upd_col_type == 0 &&
             header->update_array[l].upd_col_length == 0)  {
```

```
         strcpy (msg, "UPDATE INFORMATION NOT PROVIDED");
         PRINT (msg, -strlen(msg), 0);
         PRINT (msg, 0, 0);
         return;
      }

      col = header->update_array[l].upd_col_num;
      if (col > numColumns)  {
         sprintf (msg, "COLUMN %d NOT DEFINED", col);
         PRINT (msg, -strlen(msg), 0);
         continue;
      }
      uexit_process_data (sqlca, upd_data, &offset, col,
                          header->update_array[l].upd_col_length, &haveVAR);

   }  /* end of update array loop */
   PRINT (msg, 0, 0);

}  /* end of if  UPDATE record */

}  /* end export_sql_exit */
/
```

Chapter 4
Troubleshooting

This chapter contains notes and tips concerning running NetBase SQL Shadowing, as well as error messages which appear when the software has encountered a problem. If NetBase SQL is not performing as you expect, review the notes in this section to determine what may need to be changed. At the end of this section is a listing of the error messages associated with NetBase SQL. Listed with each message is a description and a solution. At the end of this chapter is a section on data recovery for the synchronization error situations.

Common mistakes

- **LOOPBACK** is not started. To start issue NETCONTROL START NET=LOOP at a colon prompt.

- **WSL ID** is blank or duplicated elsewhere in the system. For successful WSL, for each copy of each DBE, the **WSL ID** must be unique within the system.

- **Home_partition** is zero or duplicated elsewhere in the system. For successful WSL, each copy of each DBE must have a unique home_partition ID.

- **Max_num_partitions** is zero. For successful WSL, this value must be between 1 and 880. It should not be too large as it is multiplied by 72 to define the size of the log file for Standby Checkpoint Records. For more information, please see chapter 2.

- To insure that busy systems can continue to process without their connection with the shadow machine for a day or two without creating a hard-resync situation, define at least ten logfiles of 5,000 pages. (The maximum number of logfiles is 32)

- The SQL import process should have the highest transaction priority in case the system becomes deadlocked, since HP resolves deadlocks by aborting processes with the lowest priorities until the system is freed. (Highest priorities are the lowest numbers in the range 0-255.)

- The **SQLOUT** commands attempt to rename multiple files with the same name (but different groups and accounts). For example:

  ```
  D> SQLOUT ACCTG.DATA.PROD=DBE1.DATA.PROD,SYSB
  D> SQLOUT ACCTG.BACK.SYS=DBE2.DATA.PROD,SYSB
  ```

  In the above example, the **SQLOUT** commands would build two files named ACCTG. The import process on SYSB would not know which ACCTG file to apply where.

- **Network Transport Tables** (within NMMGR) - If many remote sessions or other Network Services are being used, some NS tables may need increasing, especially connection and socket tables.

- **Retransmission Interval Lower Bound** (within NMMGR) - It may be necessary to increase this interval if excessive network time-outs are occurring.

- The DBE was created on a much older version of ALLBASE. As a result, the DBECon file is too small to add the WSL information required to enable standby logging.

- Either the master or the shadow machine does not have two log files built. You must build at least two log files on each system involved in SQL shadowing.

Use the NMMGR program to increase these tables. If these tables have been modified, it is necessary to perform a system restart for all new values to take effect.

Termination Conditions

Conditions that result in termination of an SQL export or import process are listed below.

1. A serious error is encountered during initialization. For instance, cannot connect to DBE, error open log scan, etc.

2. The control process says to stop as a result of the user issuing a STOP command.

3. Transmit_log finds that somebody has called an WSL routine on the log file using hard resync mode. The scan is closed.

4. If the USEREXIT option is not set on the **SQLIN** record for the import process, OR the **post_sql_exit_after** routine cannot be found, AND there is an error applying the transaction.

5. The USEREXIT says to terminate.

The following conditions may cause an abort.

1. An invalid action code is returned by a USEREXIT procedure.

2. Three attempts to apply a deadlocked transaction fails.

3. Audit_log returns an error. This seems likely to happen only in the case of a program error.

4. Userexit says to disable a table but the import process can't.

NBSQLEXP may abort if it encounters:

- Cannot find the directory entry for this DBE
- DBE name is not a valid MPE file name
- The export userexit returns an invalid action code
- A hard resync point is encountered in the log scan

NBSQLEXP, pseudo node may abort if:

- USEREXIT option not specified on directory entry
- Userexit not found, must be there for pseudo node
- SCR file is invalid for this DBE
- Error opening, creating or accessing SCR file
- No partition numbers specified in directory entry

NBSQLIMP may abort if:

- Cannot find the directory entry for this DBE
- DBE name is not a valid MPE file name
- Invalid action code returned by a userexit
- Cannot open transaction save file
- Cannot disable table, action code from post-apply userexit, no pre-apply userexit

Almost all of the above errors occur during initialization. The rest are userexit errors. The following errors are the errors returned by Apply_Log that will cause NBSQLIMP to abort:

-2817	Invalid WSL log record found
-2819	WSL is not enabled
-10041	Invalid MODE specified
-10042	Invalid FLAG specified
-10044	Invalid log buffer size
-10045	Invalid size_used value
-10046	Invalid buffer offset passed
-10047	Invalid record length
-10048	Invalid BEGIN record out of seq
-10049	Modify record with no begin
-10053	Invalid # SCR slots specified
-15002	Multiple DBE in same transaction

Almost all of the above errors would occur only if something was very wrong with the import process.

Error Messages

The error messages are listed in alphabetical order. Some error messages require assistance from Quest Technical Support if they are encountered. The following list contains error messages whose solutions you may perform. If you encounter a message that is not listed below, please contact Quest Technical Support. Make a note of the error message, any details included in the error message, and the version of **NetBase** you are running.

Some of these error messages and their explanations were provided by Hewlett-Packard, and are reprinted here with permission.

ERROR MESSAGE	SUGGESTION
A BEGIN WORK WSL record encountered out of sequence. (DBERR 10048) ****Causes Abort****	Apply_Log encountered the BEGIN WORK WSL record for a new transaction while a previous Apply_Log transaction was still in progress. Probably, a single Apply_Log process is applying transactions received from multiple open scans (so the application should be checked). Transmission errors can also cause this problem. If the application is handling transactions which span multiple log buffers correctly, retry the transmission of the transactions being applied.
A WSL record in the log buffer has an invalid record length specified. (DBERR 10047) ****Causes Abort****	The Apply_Log intrinsic attempts consistency checking on the lengths (among other parameters) of the records encountered in the log buffer. If the current position in the buffer plus the length of the next log record takes it beyond the end of the used buffer space, Apply_Log will generate this error. If the BEGIN WORK op code of the next log record allows its correct length to be predetermined, the specified record length is checked against this value.
Active transactions found on partition *nnn* (DBERR 2821)	An attempt was made to hard resync a partition (through Modify_SCR) that currently has transactions active. This could also occur if the SCR array (for update) or the partition array (for delete) contained partition IDs that the user did not mean to hard resync. Verify the SCR array (or partition array) contains only partition IDs that are being hard resync'ed. If so, verify the partitions being hard resync'ed are inactive, and no transactions are active (on those partitions) before calling Modify_SCR.

**** Causes Abort **** indicates an error that aborts the apply. Other error messages (if a **post_user_exit_after** is being used) are passed to the user exit, and the apply continues.

ERROR MESSAGE	SUGGESTION

Attempting to apply a log record while WSL is not enabled. **(DBERR 2819)** ****Causes Abort****	The architecture of Apply_Log requires WSL to be active at the slave (target). Use **SQLUTIL** (**SHOWDBE**) to check the startup parameters, and activate WSL (via **START DBE NEWLOG**) if necessary.
Could not allocate heap space needed for procedure. **(DBERR 10056)**	Currently, only Open_Log_Scan can encounter this error when allocating control blocks needed for opened scans. This error simply states that the routine was unable to allocate the necessary heap space to be able to open the log scan. If the user is opening multiple scans on the same DBE, each opened scan receives a new set of control blocks (which are allocated in the heap). Check the application to see if an excessive amount of heap space is being used, and modify it to use less. If the heap allocation has previously worked on the system, check the number of users on the system, and use MONITOR to see global swap space utilization plus heap usage for individual user processes. Combining the opens into one scan will reduce the heap space required.
Duplicate HOMEPARTITION clause defined. **(DBERR 1080)**	The **START DBE NEW** command contains more than one HOMEPARTITION argument. Correct the command syntax.
Duplicate MAXPARTITIONS clause defined. **(DBERR 1081)**	The **START DBE NEW** command or the **START DBE NEWLOG** contains more than one MAXPARTITIONS keyword. Correct the command syntax.
Duplicate partition instance found in SCR array. **(DBERR 2822)**	The SCR array passed to Modify_SCR contains two rows with the same WSL_id/partition_id combination. Check the array for duplicate rows, and check the SCR information against the data returned by Get_SCR (using mode 2 - HARD resync).
Duplicate STANDBY LOGGING clause defined. **(DBERR 1082)**	The **START DBE NEW** command or the **START DBE NEWLOG** contains more than one STANDBY LOG argument. Correct the command syntax.
Duplicate WSLID clause defined. **(DBERR 1084)**	The **START DBE NEW** command contains more than one WSLID keyword. Correct the command syntax.
Hard resync record encountered in partition *nnn.* Scan aborted. **(DBERR 2815)**	An open scan found a hard resync record. This implies that a scan was not aborted on a partition that was recently hard resync'ed. Resync the partition as necessary across all soft resyncing nodes, and reopen the scan.

ERROR MESSAGE	SUGGESTION

Insufficient space in log buffer to return next log record. **(DBWARN 10077)**	Transmit log was unable to transmit the log record into the user's log buffer because the buffer did not have enough space. This may occur only if the first record to transmit will not fit. Allocate a larger log buffer space, and call transmit_log again.
Invalid buffer offset passed. **(DBERR 10046)** ****Causes Abort****	Apply_Log determined that "buffer_offset" does not point to a valid position in the user's log buffer. This happens if buffer_offset contains a negative value or a number greater than the size_used field.
Invalid FLAG specified. **(DBERR 10042)** ****Causes Abort****	The "flag" specified in the argument list is invalid. Each intrinsic has a specified set of valid flags. Correct the flag specification.
Invalid log buffer size. **(DBERR 10044)** ****Causes Abort****	The buffer size specified in the arglist is not large enough to even contain the smallest WSL record. This error can be encountered by either Transmit_Log or Apply_Log. Check the buffer size field to see if it was properly initialized. It is was, a large buffer should be defined.
Invalid log scan id passed. **(DBERR 10043)**	Either Transmit_Log or Close_Log_Scan had a "scan_id" specified in the argument list which did not match that of a valid log scan currently open by the process. Check the scan ID specified in the arglist against the scan IDs previously returned by Open_Log_Scan.
Invalid MODE specified. **(DBERR 10041)** ****Causes Abort****	The "mode" specification is invalid. Each intrinsic has a specified set of valid modes. Check the specified mode against the command specifications.
Invalid number of partitions specified. **(DBERR 10051)**	Get_SCR or Modify_SCR encountered an invalid "num_partitions" in the argument list. The values for this field are dependent on the procedure begin called in conjunction with the mode specified.
Invalid number of records in column array. **(DBERROR 10075)**	The num_columns field in the arglist has been initialized to a negative number.
Invalid number of records in update column array. **(DBERROR 10076)**	The num_upd_cols field in the argument list has been initialized to a negative number.

ERROR MESSAGE	SUGGESTION

Invalid offset for last record passed. **(DBERROR 10074)**	Audit_Log returns this message when Last_Rec_Offset contains an invalid value.
Invalid partition id encountered. **(DBERR 10052)**	Get_SCR or Modify_SCR encountered an invalid partition id passed in from the user. For Get_SCR, the partition id in error must be in the array pointed to be the "partition_array" field in the arglist. For Modify_SCR, the partition id must be in the SCR array for mode 1 (UPDATE SCR), or the partition array for mode 2 (DELETE SCR). All partition ids are required to be positive, non-zero numbers.
Invalid specification for maximum number of log files. **(DBERR 10055)**	The user specified an invalid number of "max_logfiles" for the backward scan during Open_Log_Scan. This parameter can be either -1 (the default) or a non-zero positive number.
Invalid specification for number of SCR slots used. **(DBERR 10054)**	This error can be generated by Open_Log_Scan or Modify_SCR when the "num_SCR_used" field specified in the arglist is invalid. For Open_Log_Scan and Modify_SCR (mode 1 only), "num_SCR_used" must be within the range from 1 to "num_SCR" to prevent this error.
Invalid value of used log buffer space. **(DBERR 10045)** ****Causes Abort****	Apply_Log determined that the "size_used" specified in the argument list was invalid. Either the size_used is smaller than the smallest WSL record possible, or it is greater than the number of bytes reserved in the log buffer (as indicated in the buffer_size field in the argument list). Verify the size_used field is less than or equal to the buffer_size field. Also check the size_used being passed to Apply_Log against the value that was returned by Transmit_Log.
Invalid WSL log record found. **(DBERR 2817)** ****Causes Abort****	The log record being applied in an Apply_Log was not a valid WSL log record. The log records being applied must have come from a Transmit_Log from a master instance. Check your application to verify that the log records from the Transmit_Log are being passed appropriately to the Apply_Log.
Log full	The available space in the log files is insufficient for the transaction. To correct, either perform a backup of an archive log file in "ready for backup" status (use the SQLUtil STORE command), or create an addition log file (either archive or non-archive) using the ADDLOG command.

ERROR MESSAGE	SUGGESTION

More than one slot found for the same partition (DBERR 2813)	Open_Log_Scan has received an array of SCR slots where some partition has multiple COMMIT records. To correct, retrieve a set of slots from the node performing a soft resync, by issuing a Get_SCR call with the LATEST option. Then retry the Open_Log_Scan call.
Multiple database environments updated in the same transaction. (DBERR 15002)	This can happen if a single transaction has a mixture of Apply_Log records and direct updates, or a mixture of Apply_Log records from multiple masters.
MUST BE CONFIGURED BEFORE IT CAN BE STARTED	You must modify "SQL - Enable Standby Logging" within the **CONFIG** subsystem of **NBCTRL**, setting it to "Y" prior to issuing the **START SQL** command.
New partition instance found, and the current SCR does not have any more slots. (DBERR 15001)	This typically happens on a system with a large number of partition IDs. Also, if this system has switched master-shadow states with a number of other systems, it will have a large number of non-empty partition instances. The transaction will be aborted by the ALLBASE product, but the user application will need to terminate at this point. Do a START DBE NEWLOG with a larger SCR size, and re-establish soft resync connection with the master.
No apply WSL records transaction currently active. (DBERR 10049)	Apply_Log encountered a WSL record that was not a BEGIN WORK record, and the BEGIN WORK WSL record has not been processed for the transaction. This could occur if the Apply_Log transaction was aborted and the user tried to continue applying log records without starting at the beginning of the transaction. Also, transmission errors could cause the beginning of the transaction to be lost. Check the application to verify transactions are correctly re-applied (especially in cases of transactions which span multiple log buffers). If it is correct, try to re-transmit the transaction and apply it. If the problem persists, closing and re-opening the log scan may be in order (in case the problem was due to transmission from HP SQL to the Transmit_Log process).
NO CURRENT SQL ACCESS	You must start SQL within **NBCTRL** before you may issue the **SHOW SQL** command. Enter **START SQL,** and re-issue the **SHOW SQL** command.

ERROR MESSAGE	**SUGGESTION**

No more WSL records to transmit. (DBWARN 10040)	A Transmit_Log reader has encountered the end of log. No more records can be transmitted beyond this point until some more are written. The application should pause for a short period, and then retry.
Open_Log_Scan could not reserve log for partitions found. (DBERR 2816)	The starting point for one or more partitions was overwritten by a direct update transaction. This happens during a window for which a forward scan will not lock the starting point of the attempted search. The oldest partition must be hard resync'ed, or soft resync'ed from a different system. Then you should attempt the open log scan again.
Open_Log_Scan failed on ALL specified partition. (DBERR 2814)	None of the specified partitions can be soft resync'ed. Check the resync status, and perform data recovery as necessary. If this occurred when you first attempted to enable WSL, you may have entered multiple START NEWLOG commands on the master machine. To reset the SCR, issue a START NEWLOG command without the "STANDBY LOG" argument. Then issue a complete START NEWLOG command.
Partition not found (DBWARN 2062)	A partition requested in a GET_SCR call was not found. For a GET_SCR with LATEST option call, an initial SCR slot was inserted. To correct, if the partition was wanted only if COMMITs are present, remove the partition from the request.
Some partitions (but not all) failed on Open_Log_Scan (DBWARN 2063)	In a multi-partition Open_Log_Scan, at least one partition succeeded, and at least one failed. (A scan has been opened on the successful partitions.) Check the resync status array, reviewing the return parameters for the Open_Log_Scan. Depending on the criticality of the failed partitions, the application may need to take appropriate action.
There are still some active transactions in the old SCR. (DBERR 2812)	The old SCR (out of the DBECon file) still contains some active transactions. Re-issue the **START DBE NEWLOG** command when these transactions have been completed.
Warm standby logging already disabled (DBWARN 2060)	A previous **DISABLE STANDBY LOGGING** command is still in effect. Check the sequence of operations.

ERROR MESSAGE	SUGGESTION

Warm standby logging already enabled (DBWARN 2061)	A previous **ENABLE STANDBY LOGGING** command is still in effect. Check the sequence of operations.
Warm standby logging not set for DBE (DBERR 2810)	An **ENABLE STANDBY LOGGING** command was issued for a session, but standby logging is turned off for the DBEnvironment. Either a **START DBE NEWLOG** needs to be issued to start standby logging for the DBE, or the **ENABLE STANDBY LOGGING** is invalid.
WSLID too long (DBERR 1085)	The WSL ID was more than 8 characters. Correct the command, and re-issue it.

SQL ERRORS

The export and the import processes may incur errors when reading records from the log file or when updating the shadow copy of the DBE. These errors are printed to the console using SQLEXPLAIN. A number of errors that may occur on the shadow machine are not recoverable. In general, these are errors that indicate that the DBEs are out of sync. If the process receives one of these errors, it prints a message and terminates.

ERROR RECOVERY

If a machine that is exporting or importing updates gets a datacomm failure, the SQL processes will stay up and will attempt to reestablish communications with the remote node.

If a machine crashes, much the same is true. The SQL processes on the other machines will stay up attempting to reestablish a connection. The machine that crashed will be restarted, NetBase will be restarted, and NetBase will automatically start up the SQL processes. These processes will connect to the remote nodes and SQL shadowing will continue.

In either case updates cannot be lost because the way that WSL works insures that shadowing starts with the last complete update applied on the shadow machine. SQL is secure enough that the system crash is unlikely to hurt the DBE.

If a DBE does become corrupt, or the disc fails, or the DBEs go out of sync, then the DBEs must be resync'ed. The procedures for doing this depend on the type of failure and on the particular system design. In general, what is required is to unload all the tables that make up the out-of-sync partition from the master copy of the DBE. At the same time the checkpoint record must be retrieved from the master copy's WSL log file. Then those same tables are loaded with the data from the master copy before the logfile is updated with the checkpoint record from the master.

Remember that the partition number is a logical construct. Nothing in SQL or WSL associates a particular table with a particular partition number. Speaking in those terms assumes that no two master machines update the same table.

Resync’ing

As an example, let's start with the simplest case. A shadow copy of a DBE is out of sync with the master. There is only one shadow copy and one master copy. To get back into sync the following steps should be taken:

1. Connect to the DBE on the master system.

2. Unload the DBE. This should be done while the DBE is completely quiesced. Do not proceed until all activity with the DBE has ceased.

3. Get the checkpoint record from the log file. It is important that no other activity takes place on the DBE between the start of step 2 and this step. Otherwise the DBEs will be out of sync at the completion of this process.

   To get the checkpoint record, you may use the **NBSQLSYN** utility.

   ```
   : RUN NBSQLSYN.NB.NETBASE;INFO="GET,dbename,partition"
   ```

   Specify the partition from which the data is being resync'ed (the home partition number of the master). The results are placed in a file named **SYNCSCR** in the same location as the DBE. So if the DBE is in DATA.PROD, the file will be SYNCSCR.DATA.PROD.

4. Connect to the DBE on the shadow system.

5. Again, make sure that the DBE is completely quiesced. Then delete the old data from the DBE and load the tables with the data from the master DBE.

6. Update the checkpoint in the local log file from the checkpoint record retrieved from the log file on the master machine.

   To update the SCR using the **SYNCSCR** file, transfer it to the shadow machine. With that accomplished, you may use the **NBSQLSYN** utility again.

   ```
   : RUN NBSQLSYN.NB.NETBASE;INFO="UPDATE,dbename"
   ```

The whole DBE need not necessarily be replaced. Consider an installation that has three master machines and one slave. In this case we may assume that each master only updates a particular set of tables. For instance, the master on SYSA updates tables 1-3, the master on SYSB updates tables 4-7, and the master on SYSC updates tables 8-9. In effect, the shadow machine acts as the shadow copy for three different DBEs. If the shadow machine goes out of sync with the database on SYSC, then the above steps are followed but only for tables 8-9.

If userexits are used to filter the data from the master copy, a simple unload and reload will not suffice. In this case the best plan would be to restore the shadow copy to the last backup and then restart SQL shadowing. The backups should always include the logfiles so that when SQL shadowing is started, the checkpoint record on the shadow machine is a correct reflection of the updates on that machine.

NetBase SQL Shadowing

Index