Cognos(R)
Application Development Tools
PowerHouse(R) 4GL

VERSION 8.4E

POWERHOUSE AND RELATIONAL DATABASES

COGNOS®

THE NEXT LEVEL OF PERFORMANCE™

Product Information

This document applies to PowerHouse(R) 4GL Version 8.4E and may also apply to subsequent releases. To check for newer versions of this document, visit the Cognos support Web site (http://support.cognos.com).

Copyright

Table of Contents

About this Book

Overview

Chapter 1, "About PowerHouse and Relational Databases", provides an overview of PowerHouse support for relational databases that are identified in your dictionary.

Chapter 2, "Relational Support in QDESIGN", provides information about QUICK transaction models, overriding the transaction defaults in QUICK, attaches and transactions in QUICK, tuning attaches in PowerHouse, and transaction error handling in QUICK.

Chapter 3, "Relational Support in QTP", provides information about QTP transaction models, overriding the transaction defaults in QTP, attaches and transactions in QTP, tuning attaches in PowerHouse, and transaction error handling in QTP.

Conventions in this Book

This book is for use with MPE/iX, OpenVMS, UNIX, and Windows operating systems. Any differences in procedures, commands, or examples are clearly labeled.

In this book, words shown in uppercase type are keywords (for example, SAVE). Words shown in lowercase type are general terms that describe what you should enter (for example, filespec). When you enter code, however, you may use uppercase, lowercase, or mixed case type.

Getting Help

For more information about using this product or for technical assistance, visit the Cognos Global Customer Services Web site (http://support.cognos.com). This site provides product information, services, user forums, and a knowledge base of documentation and multimedia materials. To create a case, contact a support person, or provide feedback, click the **Contact Us** link at the bottom of the page. To create a Web account, click the **Web Login & Contacts** link. For information about education and training, click the **Training** link.

Cognos PowerHouse 4GL Documentation Set

PowerHouse 4GL documentation includes planning and configuration advice, detailed information about statements and procedures, installation instructions, and last minute product information.

Objective	Document
Install PowerHouse 4GL	*Cognos PowerHouse 4GL & PowerHouse Web Getting Started* book. This document provides step-by-step instructions on installing and licensing PowerHouse 4GL. Available in the release package or from the following website: http://support.cognos.com

Objective	Document
Review changes and new features	*Cognos PowerHouse 4GL & PowerHouse Web Release and Install Notes*. This document provides information on supported environments, changes, and new features for the current version. Available in the release package or from the following website: http://support.cognos.com
Get an introduction to PowerHouse 4GL	*Cognos PowerHouse 4GL Primer*. This document provides an overview of the PowerHouse language and a hands-on demonstration of how to use PowerHouse. Available from the PowerHouse 4GL documentation CD or from the following website: http://powerhouse.cognos.com
Get detailed reference information for PowerHouse 4GL	Cognos PowerHouse 4GL Reference documents. They provide detailed information about PowerHouse rules and each PowerHouse component. The documents are • *Cognos PowerHouse 4GL PowerHouse Rules* • *Cognos PowerHouse 4GL PDL and Utilities Reference* • *Cognos PowerHouse 4GL PHD Reference* • *Cognos PowerHouse 4GL PowerHouse and Relational Databases* • *Cognos PowerHouse 4GL QDESIGN Reference* • *Cognos PowerHouse 4GL QUIZ Reference* • *Cognos PowerHouse 4GL QTP Reference* Available from the PowerHouse 4GL documentation CD or from the following websites: http://support.cognos.com and http://powerhouse.cognos.com

Cognos PowerHouse Web Documentation Set

PowerHouse Web documentation includes planning and configuration advice, detailed information about statements and procedures, installation instructions, and last minute product information.

Objective	Document
Start using PowerHouse Web	*Cognos PowerHouse Web Planning and Configuration book*. This document introduces PowerHouse Web, provides planning information and explains how to configure the PowerHouse Web components. **Important:** This document should be the starting point for all PowerHouse Web users. Also available from the PowerHouse Web Administrator CD or from the following websites: http://support.cognos.com and http://powerhouse.cognos.com
Install PowerHouse Web	*Cognos PowerHouse 4GL & PowerHouse Web Getting Started* book. This document provides step-by-step instructions on installing and licensing PowerHouse Web. Available in the release package or from the following website: http://support.cognos.com
Review changes and new features	*Cognos PowerHouse 4GL & PowerHouse Web Release and Install Notes*. This document provides information on supported environments, changes, and new features for the current version. Available in the release package or from the following website: http://support.cognos.com
Get detailed information for developing PowerHouse Web applications	*Cognos PowerHouse Web Developer's Guide*. This document provides detailed reference material for application developers. Available from the Administrator CD or from the following websites: http://support.cognos.com and http://powerhouse.cognos.com
Administer PowerHouse Web	The *PowerHouse Web Administrator Online Help*. This online resource provides detailed reference material to help you during PowerHouse Web configuration. Available from within the PowerHouse Web Administrator.

Cognos Axiant 4GL Documentation Set

Axiant 4GL documentation includes planning and configuration advice, detailed information about statements and procedures, installation instructions, and last minute product information.

Objective	Document
Install Axiant 4GL	*Cognos Axiant 4GL Web Getting Started* book. This document provides step-by-step instructions on installing and licensing Axiant 4GL. Available in the release package or from the following website: http://support.cognos.com
Review changes and new features	*Cognos Axiant 4GL Release and Install Notes*. This document provides information on supported environments, changes, and new features for the current version. Available in the release package or from the following website: http://support.cognos.com
Get an introduction to Axiant 4GL	*A Guided Tour of Axiant 4GL*. This document contains hands-on tutorials that introduce the Axiant 4GL migration process and screen customization. Available from the Axiant 4GL CD or from the following websites: http://support.cognos.com and http://powerhouse.cognos.com
Get detailed reference information on Axiant 4GL	*Axiant 4GL Online Help*. This online resource is a detailed reference guide to Axiant 4GL. Available from within Axiant 4GL or from the following websites: http://support.cognos.com and http://powerhouse.cognos.com

For More Information

For information on the supported environments for your specific platform, as well as last-minute product information or corrections to the documentation, see the *Release and Install Notes*.

Chapter 1: About PowerHouse and Relational Databases

Overview

This chapter provides an overview of PowerHouse support for relational databases that are identified in your dictionary. You'll find information about

- the PowerHouse environment
- item and element attributes
- transactions, transaction types, and transactions in QUIZ
- auditing database operations
- database restructuring considerations
- trouble-shooting relational access problems
- PowerHouse SQL statements
- PowerHouse SQL architecture
- cursors in PowerHouse, customizing cursors, and linking cursors
- developer-written SQL queries
- stored procedures and RDBMS-specifics
- creating user-defined functions (**DB2, Oracle**)

About PowerHouse and Relational Databases

PowerHouse uses transactions and transaction control features to define how your PowerHouse application and relational database work together. Transactions define the interactions between an application and a database. Transaction control governs how transactions are started, committed, or rolled back within an application.

PowerHouse supports the following relational databases:

Platform	Supported Relational Database Type
MPE/iX	ALLBASE/SQL
OpenVMS:	Oracle, Oracle Rdb (as RDB and RDB/VMS)
UNIX, Windows:	DB2, ODBC (also used for Microsoft SQL Server), Oracle, Sybase

PowerHouse uses the facilities of each specific database product to implement a transaction model based on PowerHouse's defaults and the specifications of the application designer. Wherever possible, this model is consistent for all database products. Where databases differ (for example, in the area of transaction management strategies and options), the transaction attributes supported by QUICK and the database determine the behavior of the transaction model.

Database specific differences are addressed in this chapter. Please ignore those that do not apply to your environment.

The PowerHouse Environment

PowerHouse treats a relational database's catalog (or metadata) as a subdictionary of a PowerHouse dictionary. All PowerHouse components can access the information in a database that has been identified to a PowerHouse dictionary. QUICK and QTP can read data from and write data to relational databases. QUIZ can read data from databases to generate reports. PowerHouse cannot define or create relational databases. You must use the database administrative tools for your database to create and maintain the database.

Identifying a Relational Database to PowerHouse

You use the DATABASE statement in PDL to identify a relational database to PowerHouse. For more information on identifying a relational database to PowerHouse, see the DATABASE statement, in Chapter 2, "PDL Statements", in the *PDL and Utilities Reference* book.

The Impact of Case Sensitivity when Identifying Databases

ALLBASE/SQL and Sybase names are case-sensitive. PowerHouse upshifts all components of table and column names by default. Unless this information is in uppercase in your database, you should use the **downshift** or **noshift** program parameter and/or the DOWNSHIFT or NOSHIFT options of the SET statement to ensure that PowerHouse can locate the data you want.

Referencing Tables, Columns, and Views in a Database

Once a database has been identified to PowerHouse, you can access the tables, columns, and views it contains. The steps PowerHouse takes to locate the database containing a table differ depending on whether the statement is an SQL or non-SQL statement. PowerHouse searches for a table using the following steps.

	For each SQL Statement	For each non-SQL statement
1.	PowerHouse uses the "IN database" option on the SQL statement. The database name must be the name of a database identified to the current dictionary.	PowerHouse uses the "IN database" option on the FILE statement. The database name must be the name of a database identified to the current dictionary.
2.	If there is no "IN database" option, PowerHouse uses the database specified in the SET DATABASE statement that is in effect.	If there is no "IN database" option, PowerHouse uses any matching record definitions it finds in the dictionary files.
3.	If there is no SET DATABASE statement, PowerHouse uses the database specified in the resource file.	If no matching record definitions are found, and the **subdict=search** program parameter is set, PowerHouse searches all databases identified to the dictionary.
4.	If there is no database specified in the resource file, PowerHouse sets the default to the name of the first DATABASE statement specified in the dictionary. If the order of DATABASE statements in the dictionary is changed, then the default changes.	

PowerHouse issues an error message if the database containing the table or view can't be located.

PowerHouse Security

PowerHouse assumes that the database controls data security. PowerHouse data security is not checked for any database if access is via SQL statements. PowerHouse application security is still enforced, for example, if the USERS INCLUDE option on the SCREEN statement is used.

For databases, you can assign PowerHouse application security classes to elements in order to augment database security for access via non-SQL statements. For more information, see the APPLICATION SECURITY CLASS statement in Chapter 2, "PDL Statements", in the *PDL and Utilities Reference* book.

PowerHouse allows access to system tables, assuming you have database access privileges to them. However, SHOW statements do not show information on system tables. You will have to refer to database documentation to see what system tables are present.

Item and Element Attributes

PowerHouse automatically retrieves all of what it considers to be item attributes for a column directly from the relational database.

PowerHouse cannot retrieve all of the element information it requires from a relational database because there are differences between the element attributes that a dictionary provides and the element information that a relational database provides. PowerHouse retrieves element attributes for a column in a relational table as follows:

1. First, PowerHouse searches the current dictionary for an element whose name and basic datatype match the column name and basic datatype. The basic datatypes are character and numeric; dates are considered to be numeric. If there is such an element, PowerHouse uses that element's attributes.
2. If there isn't a matching element in the dictionary, PowerHouse generates default element attributes for the column based on the attributes that are available for the column from the relational database.

When PowerHouse generates default element attributes, it is possible for two columns that have the same name, but are in different tables, to have different values for some attributes. For example, if they have different sizes, their default pictures differ. To avoid such discrepancies, enter element definitions for the fields into the dictionary.

If you have a relational database with a date column and an element with the same name but with different characteristics (particularly century included versus century excluded), PowerHouse may display the dates incorrectly. You must ensure that your element characteristics are compatible with your column definition.

If you create a dictionary element definition for a relational column, you should set the input scale, output scale, and decimal positions in the same way PowerHouse would set them by default. This ensures that the relational system you are using and PowerHouse have consistent views of the columns.

Decimal Positions

The default number of decimal places depends on the datatype of the column, its scale factor and the element size. SQL supports the scale factor as an optional requirement on decimal and numeric datatypes; a positive integer represents the number of decimal places.

RDB/VMS allows a scale option on all numeric datatypes; but a negative integer represents the number of decimal positions (for example, SCALE -2 means that the number will be scaled on output by 10-2). The default chosen by the dictionary will always be the lesser of the scale factor (ignoring the negative sign) and the element size. If the field scale factor in an RDB/VMS database is positive, then zero is used as the default decimal position.

Input Scale

For SQL, the input scale defaults to the column scale factor. For RDB/VMS, the input scale defaults to the absolute value of the field scale factor, if negative. Otherwise, it defaults to the number of decimal places minus the output scale.

Output Scale

For SQL, the output scale defaults to zero. For RDB/VMS, the output scale defaults to the field scale factor if it is positive. Otherwise, it defaults to the number of decimal positions minus the input scale.

VARCHARS

In QTP, if you use this syntax

```
TEMP x VARCHAR SIZE 6
```

it is assumed that you mean that the physical size should be 6 bytes, and therefore, you can only put 4 bytes of data into the TEMP, leaving 2 bytes for the string length. If you are mixing TEMPs (to be written to a subfile) and database VARCHAR fields, use the following code instead to avoid confusion:

```
TEMP x VARCHAR*6
```

where, for example, 6 is the size of the string in the database.

In this case, 6 is the element size and PowerHouse will add 2 to determine the default physical size for the item. Thus the TEMP and the Database item will be the same size in the subfile and can be used interchangeably. QSHOW will display the ITEM size, which would be 8 in this case.

SQL Date and Time Expressions

If you are using date or time expressions inside SQL (such as assigning a date value within an SQL INSERT), you are required to prefix the expression with either "date", "time", "datetime", or "timestamp". For example,

```
date '1980-05-06'
time '09:23:21.000'
datetime '1980-05-06 09:23:21.000'
timestamp '1978-09-01 01:52:27.000'
```

If you do not prefix the date or time expressions with "date", "time", "datetime", or "timestamp", then the following error message appears:

```
EXPENG-E-TYPE1, The operation <litstr> is invalid for data type <null>.
```

Transaction Overview

PowerHouse uses transactions to access and manipulate data in relational databases. A transaction is a set of tasks which are either all completed, or if any fail, are all undone. Transactions have three main roles:

- to define a "unit of work"
- to provide a consistent picture of the database for each user
- to control update conflicts among concurrent database users

With all file systems and databases, you can define PowerHouse transactions by passing and receiving records and tables between screens. A transaction may start on one screen, continue through several screens, and end on another.

With a relational database, you can consider many other definitions of a PowerHouse transaction that go beyond the options supported by non-relational systems. You may decide that several activities - perhaps involving several update actions - should be grouped together and treated as a single PowerHouse transaction that is either committed or rolled back completely.

Transaction Types

In PowerHouse, there are three types of transactions: database transactions, PowerHouse transactions, and conceptual transactions.

Database Transaction

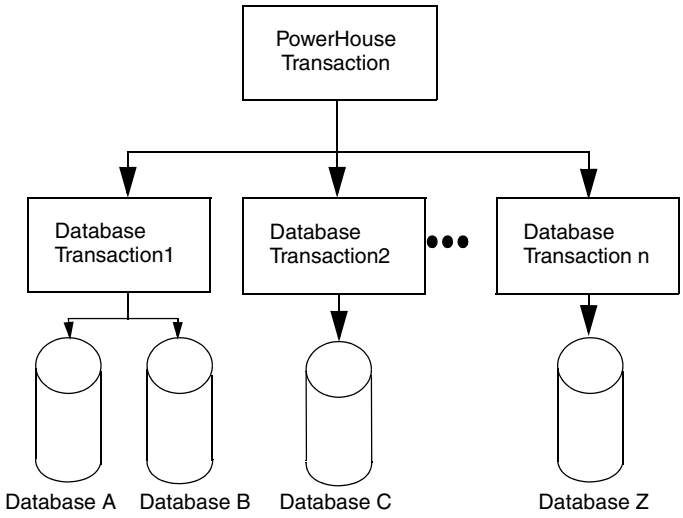
A database transaction is a unit of work known to the relational database management system. A database transaction can be used to access one or more different databases and, indirectly through gateways, different database types. The relational database management system maintains the integrity of these transactions.

PowerHouse Transaction

A PowerHouse transaction consists of one or more database transactions managed by PowerHouse as a single unit. The underlying database transactions in a PowerHouse transaction may be started at different times by PowerHouse. When PowerHouse commits or rolls back the PowerHouse transaction, all of the underlying database transactions are either committed or rolled back at the same time. A database transaction can only be associated with one PowerHouse transaction.

PowerHouse defines and manages transactions through transaction models. Each of the PowerHouse components, QUIZ, QDESIGN, and QTP, uses a transaction model. For information about the QUIZ transaction model, see (p. 15). For information about the QDESIGN transaction model, see Chapter 2, "Relational Support in QDESIGN". For information about transactions in QTP, see Chapter 3, "Relational Support in QTP".

PowerHouse transactions can be defined using the TRANSACTION statement and may be related to tables using the TRANSACTION option of the QDESIGN FILE and CURSOR statements, and SQL DML verbs. PowerHouse maintains the integrity of these transactions.



Conceptual Transaction

A conceptual transaction is one or more PowerHouse transactions which span QUICK screen boundaries that the screen designer views as a related group of operations. The screen designer must maintain the integrity of these transactions.

Transactions and Threads

Transactions are not shared between screen threads.

Transactions in QUIZ

QUIZ has a single read-only transaction that is started at the beginning of the report and is committed at the end of the report.

Default Transactions in QUIZ

The QUIZ default transaction names and attributes are:

```
TRANSACTION QUIZ_QUERY  READ ONLY READ COMMITTED
TRANSACTION QUERY       READ ONLY READ COMMITTED
```

Overriding the Transaction Defaults in QUIZ

The behavior of the transaction that QUIZ uses may be customized. In QUIZ, the attributes for the Query transaction are determined as follows:

1. QUIZ sets the attributes by looking in the dictionary for a transaction named QUIZ_QUERY.
2. If a QUIZ_QUERY transaction has not been defined in PDL, then QUIZ sets the attributes by looking in the dictionary for a transaction named QUERY.
3. If there is no QUIZ_QUERY or QUERY transaction defined in the dictionary, then the transaction name defaults to QUERY and the attributes are set to the default values specified for the options of the PDL TRANSACTION statement.

Auditing Database Operations in PowerHouse

You can audit certain database operations when running your PowerHouse application using the **dbaudit** program parameter. The information is reported on your default output device. The following database operations are audited:

- attaches
- detaches
- request compiles
- request releases
- request starts
- transaction commits
- transaction prepares
- transaction rollbacks
- transaction starts

Transaction activity is common to database transactions and PowerHouse transactions and both are audited.

In the audit you can see that for each

- attach there is one or more PowerHouse transactions
- PowerHouse transaction there is one or more database transactions
- database transaction there is one or more request compiles, request starts, and request releases. A request may be re-used several times.

Each attach, database transaction, and request is uniquely identified by a number called a handle. The handle is reported as part of the audit to associate requests with transactions and transactions with attaches.

For details on the **dbaudit** program parameter options, see the section "dbaudit", in Chapter 2, "Program Parameters", in the *PowerHouse Rules book*.

Database Restructuring and Your PowerHouse Application

Relational databases can be dynamically restructured. New tables, columns, or indexes can be added to an existing database. In general, PowerHouse source programs (like QUIZ reports and QDESIGN source) are not affected by database structure changes unless they reference a table or column that has been dropped from the database, or when columns in a table have been reordered.

The following structure changes do not generally affect compiled PowerHouse programs:

- new tables or views added to the databases
- new columns added to a table if the column can be NULL or has a default assigned by the database
- tables or columns that have been dropped and that were not referenced in the program
- indexes that are dropped (unless referenced in a VIAINDEX option)
- indexes that are deactivated (unless referenced in a VIAINDEX option)
- triggers that perform no updates are added or modified

These structure changes affect QUICK screens. Recompilation and/or modifications may be necessary:

- a column is made updatable; or is made not updatable
- columns that cannot be NULL are added
- a view is made updatable
- a column datatype or size is changed
- columns of a table have been reordered
- an index is changed from Unique to Repeating, or vice versa
- triggers that update the database are added or modified (this may cause side-effects that necessitate redesign)
- indexes are added or dropped because this affects the default access paths generated by PowerHouse. A specific reference to an index name within a VIAINDEX clause will be invalid if the index has been dropped. (Use of the VIA option is preferred over VIAINDEX for this reason.)

Changes to element definitions in the PowerHouse dictionary may also be required, particularly if column datatype or size changes have been made, or if you want to keep element validation rules synchronized with changes to column validation rules.

Troubleshooting Relational Access Problems in PowerHouse

If you encounter problems accessing a relational database with PowerHouse, run through the following checklist to try and isolate the problem:

- check that you're using the correct PowerHouse dictionary
- check that the database is defined correctly in the data dictionary (for example, verify that it has the correct open name)
- make sure that you qualify your table names with the IN option, or use the **subdict=search** program parameter, the SUBDICTIONARY resource file statement, or the SET DATABASE statement for SQL
- ensure that the table you are accessing is from a relational database and is not a non-relational file that has the same name
- try accessing the database using utilities provided by the database vendor
- use Debugger, the **dbaudit** program parameter, and/or the SET LIST SQL statement for SQL to determine when the problem occurs

SQL Overview

This section and the sections that follow cover topics related to using SQL in PowerHouse. They are intended for experienced users of SQL and relational databases. For more information about standard SQL syntax and language rules, refer to your SQL documentation.

SQL in PowerHouse is a set of statements and verbs that you use the same way as other PowerHouse statements and verbs. SQL statements and existing statements affected by SQL are as follows:

PDL	DATABASE, FILE

QDESIGN Statements	ACCESS, CURSOR, SQL DECLARE CURSOR (query specification and stored procedure), FIELD, SET
QDESIGN Procedures and Verbs	FIND, PATH, SQL CALL, SQL CLOSE, SQL DELETE, SQL FETCH, SQL INSERT, SQL OPEN, PUT, SQL UPDATE, WHILE RETRIEVING
QUIZ	ACCESS, CHOOSE, SQL DECLARE CURSOR (query specification and stored procedure), SET
QTP	ACCESS, SQL CALL, CHOOSE, SQL DECLARE CURSOR (query specification and stored procedure), SQL DELETE, EDIT, SQL INSERT, SET, SQL UPDATE

SQL statements generate SQL code which is passed to the database for processing. SQL verbs can be used within procedures or within components to act directly on a database. All SQL verbs in generated code are preceded by the SQL keyword.

SQL statements in PowerHouse applications can be divided into two categories: those that issue direct data manipulation requests, and those that define and manage cursors. Direct data manipulation statements are SQL INSERT, SQL UPDATE, and SQL DELETE. These are available as procedural verbs in QDESIGN and as statements in QTP.

The use of cursors in PowerHouse is discussed in detail later in this chapter. For further information on each of the statements and verbs affected by SQL, see the appropriate reference book.

SQL Architecture

In PowerHouse, SQL is supported through an SQL engine in the data access layer of PowerHouse. This engine consists of two portions: a relational access layer that processes all SQL statements and is common for all supported databases, and a number of database-specific access modules.

If an SQL statement contains functions or operations that are not available in the underlying database, then the access layer sends a database-specific request for the portion of the request that is supported, and processes the remainder of the request itself. For example, if the SQL request contains a PowerHouse function, then that PowerHouse function is evaluated by PowerHouse, and the remainder of the request is processed by the database.

For example, if the PowerHouse SQL statement is

```
>   DECLARE X CURSOR FOR SELECT &
>   UPSHIFT(LAST_NAME), SKILL FROM EMPLOYEES, SKILLS
>       WHERE...
```

then the request sent to the database would be

```
SELECT LAST_NAME, SKILL FROM EMPLOYEES, SKILLS WHERE ...
```

The access layer would take care of applying the UPSHIFT function to the LAST_NAME data returned by the database.

In general, features that are beyond SQL92 Entry Level are not supported by many databases and therefore are processed within the access layer. These features include:

- case-expressions
- datetime expressions (if not supported by the database)
- derived tables
- the COUNT ALL aggregate
- SQL data manipulation functions
- most outer join specifications
- some types of joined tables (when it is used to enforce join order).

In addition, references to PowerHouse functions in SQL statements will be processed within the access layer, since the ability to use these functions within SQL statements is an extension provided by Cognos.

The SQL data manipulation functions are listed in Chapter 6, "Functions in PowerHouse", in the *PowerHouse Rules book*. They are identified by the code SQL-DMF.

SQL 92 Compatibility

PowerHouse 4GL provides strict SQL 92 compatibility. In many cases, this removes ambiguity and differences between databases. However, in some cases, it means that code from previous releases causes parse errors. To remove the requirement for strict SQL 92 compatibility, use the STRICT_SQL92 environment variable and set the value to NO. The default value is YES.

Quoted Stored Procedure Calls

As part of the strict SQL 92 compatibility mentioned above, quoted stored procedure calls, where the quoted procedure call syntax is passed directly to the database, cause parse errors. For example:

```
> DECLARE mycursor CURSOR FOR CALL "myproc('param')"
```

is not accepted. In order to allow the double quotes and pass what is between the double quotes directly to the database, specify the **quotedproccall** program parameter at compile time.

Viewing Generated SQL Code

To view the SQL code that PowerHouse prepares for the database, use SET LIST SQL. In the following example, options of the SQL DECLARE CURSOR and the CURSOR statements are combined when the SQL code is generated. SET LIST SQL shows the resulting SQL query that will be used to retrieve data from the database. The code preceded by ___ is displayed when SET LIST SQL is used.

```
>   SET LIST SQL
>   SQL DECLARE EMPLIST CURSOR FOR &
>   SELECT * FROM EMPLOYEES, BRANCHES
>   SCREEN EMPBRANCHC
>   CURSOR EMPLIST &
>   WHERE (EMPLOYEES.BRANCH = BRANCHES.BRANCH) &
>   PRIMARY KEY EMPLOYEE
__  Sql after substitutions are applied:
__  SELECT *
__     FROM EMPLOYEES, BRANCHES
__     where EMPLOYEES.BRANCH = BRANCHES.BRANCH
__  Sql after PowerHouse variables/expression are removed:
__  SELECT *
__     FROM EMPLOYEES, BRANCHES
__     where EMPLOYEES.BRANCH = BRANCHES.BRANCH
>   FIELD EMPLOYEE OF EMPLIST
>   FIELD FIRST_NAME OF EMPLIST
>   FIELD LAST_NAME OF EMPLIST
.
.
.
```

Resetting Bind Variables in SQL Statements

A bind variable is a placeholder in SQL generated at compile time where a value will be substituted at execution time. For example, if a request value for a Find is needed in generated SQL, a bind variable acts as the placeholder in the WHERE clause. Each bind variable has a unique identifier made up of a number and the field name. In PowerHouse versions previous to 8.4xD1, the number was incremented from statement to statement even though the field was the same. This meant that generated SQL was different even though the SQL statements themselves were the same. Because the generated SQL was different, it could not be reused by the database.

As of 8.4xD1, the **resetbindvar** program parameter specifies that the bind variables are to be reset for each SQL statement. This allows the generated SQL to be identical for identical SQL syntax. The bind variables will be a letter and a number. The letter is S for Select operations, U for update operations, I for insert operations, and D for delete operations. The number is incremented from 1.

The program parameter is available in QDESIGN, QUICK, QUIZ, and QTP. The default is **resetbindvar**. To override the default, use **noresetbindvar**. There is also a resource file statement RESETBINDVAR|NORESETBINDVAR.

Setting the Database

Each SQL statement in PowerHouse is associated with a specific database defined in the PowerHouse dictionary.

You specify the database using either the SET DATABASE statement or the IN database option of SQL statements. The database name that you use must be the name of a database defined in the current dictionary.

If you don't specify the IN database option, PowerHouse takes the default from the SET DATABASE statement. If there is no SET DATABASE statement, PowerHouse takes the default from the resource file. If there is no resource file entry, PowerHouse sets the default to the name of the first DATABASE statement that appears in the dictionary code. If the order of the dictionary code is changed, the default changes.

Using Program Variables in SQL

Program variables within SQL statements indicate references to PowerHouse variables, rather than database column names. The colon (:) is used before the name to indicate that it is a PowerHouse variable reference. The reference can be a PowerHouse item or an expression within parentheses.

Program variables are often used to:

- provide values for selection conditions
- provide the column values for new rows (in INSERT statements) or changed columns (in UPDATE statements)

In the following example, DEPT is the name of a column in the EMPLOYEES table, whereas :DEPTNO is the name of a locally defined program variable.

```
>   DELETE FROM EMPLOYEES WHERE DEPT = :DEPTNO
```

In the following example, the value of :PROJID is provided by the application, and can be changed each time the cursor is opened:

```
>   DECLARE GETAPROJECT CURSOR FOR &
>       SELECT * FROM PROJECTDETAILS &
>       WHERE PROJECTID = :PROJID &
>       ORDER BY PROJECTDATE
```

This allows the application to provide values that should be used in data selection conditions, but provides fairly limited flexibility for any interactive application. For example, the field (PROJECTID) that identifies the rows to be selected cannot be changed, and the order of data returned also cannot be changed. To retrieve the same information in a different order, a second cursor declaration is required.

Cursors in PowerHouse

SQL relies on a cursor to manage the rows returned from the database. A cursor is a mechanism that enables application programs to work with the rows of columns (and/or derived columns) satisfying a particular query specification.

You define a cursor in PowerHouse using the DECLARE CURSOR statement. The DECLARE CURSOR statement specifies the query the application wants to send to the database. In many ways a cursor declaration is like a view definition, except that a cursor is an object within an application, while a view is an object within a database. A declared cursor has a name to identify it, but does not have any local storage associated with it. Opening a cursor executes the query specification, making the results available to the application.

When retrieving data from multiple tables in the same database, you should use the WHERE option or the JOIN...ON option of the query specification to define how the tables are related.

The following is an example of a cursor declaration in PowerHouse. All examples in this chapter assume upper case names in databases. SQL statements in PowerHouse are preceded by the SQL tag and use the PowerHouse continuation character (&).

```
>   SQL DECLARE STUDENTLIST CURSOR FOR &
>       SELECT LASTNAME, FIRSTNAME, COURSE, GRADE, &
>       (DURATION * HOURS) AS CREDITHOURS &
>       FROM STUDENTS, GRADES &
>   WHERE STUDENTS.STUDENTID = GRADES.STUDENTID &
>       AND GRADES.SEMESTER LIKE '92%' &
>   ORDER BY LASTNAME, FIRSTNAME
```

The scope of a cursor declaration is determined by the location of the cursor. For example, in QTP a cursor's scope may be a single request, an entire run, or an entire QTP session. Similarly, in QUIZ and QDESIGN, the scope of a cursor's declaration may extend beyond a single report or screen definition. Within its scope, a cursor name must be unique.

A cursor refers to a single database. To access information from multiple databases, use two or more DECLARE CURSOR statements and link the resulting cursors. For more information about linking cursors, see (p. 26).

Using Cursors in QDESIGN

You use the DECLARE CURSOR statement to specify the query that the application sends to the database. You use the CURSOR statement to create local storage to hold the results after execution of the query and indicate the role of the cursor within the screen. The CURSOR statement can also refer to a table or view in a database.

The scope of any cursor in QDESIGN is determined by when you declare it.

- If you declare a cursor in the data section of a screen, it is valid until the next BUILD or CANCEL statement.
- If you declare a cursor before a SCREEN statement, it stays valid for the remainder of the QDESIGN session or until the next CANCEL statement.

For multiple table cursors, only the columns belonging to the first table after the first FROM keyword in the query can be updated by default. If you use the GENERATE statement, the FIELD statements for all other columns have the DISPLAY option.

Using Cursors in QUIZ

In QUIZ, cursor declarations identify the data retrieved from the database for further processing. Cursor declarations during a QUIZ session remain valid until a CANCEL command is issued.

You use the ACCESS statement in QUIZ to identify and relate data from various sources. The ACCESS statement may refer to one or more declared cursors as well as subfiles and/or files declared in the dictionary.

For example:

```
>   DECLARE EMPREPORT CURSOR FOR &
>       SELECT LASTNAME, FIRSTNAME, PROJECTNAME,&
>              BILLINGAMOUNT &
>       FROM EMPLOYEES, PROJECTS, BILLINGS &
```

```
>      WHERE EMPLOYEES.EMPID = BILLINGS.EMPID &
>         AND BILLINGS.PROJECTID = PROJECTS.PROJECTID &
>         AND PROJECTS.STATUS = 'A'
>
>   ACCESS EMPREPORT
>
>   REPORT ALL
>   GO
```

Cursor declarations can get quite sophisticated, and you can use all the features of QUIZ on the data retrieved using the cursor.

The following example illustrates a cursor that returns only the aggregate data from the database (that is, one row of summary data per month):

```
>   SQL DECLARE JUSTTOTALS CURSOR FOR &
>      SELECT ORDERMONTH, COUNT(*) AS KOUNT, &
>         SUM(QUANTITY) AS SUMQUANTITY &
>         FROM LOTSADETAILS &
>            GROUP BY ORDERMONTH
```

The following example incorporates a subquery in the WHERE clause in order to select project numbers that have 5 or more employees allocated:

```
>   SQL DECLARE VALIDPROJECTS CURSOR FOR &
>      SELECT PROJECTID FROM PROJECTS &
>   WHERE 5 <= (SELECT COUNT(DISTINCT EMPID) &
>         FROM BILLINGS &
>         WHERE BILLINGS.PROJECTID = PROJECTS.PROJECTID)
```

A cursor refers to a single database. To access information from multiple databases, link two or more cursors using the ACCESS statement in QUIZ and QTP. In the following example:

- The OVERBUDGET cursor selects data from the FINANCE database.
- The PERSINFO cursor accesses data from the PERSONNEL database.

```
>   SQL IN FINANCE DECLARE OVERBUDGET CURSOR FOR &
>      SELECT MANAGERID, PROJECTID, FORECAST, ACTUAL &
>      FROM PROJECTS &
>      WHERE ACTUAL > (FORECAST * 1.10)
>
>   SQL IN PERSONNEL DECLARE PERSINFO CURSOR FOR &
>      SELECT EMPLOYEEID, LASTNAME, FIRSTNAME &
>      FROM EMPLOYEES &
>      WHERE EMPLOYEEID = :MANAGERID
>
>   ACCESS OVERBUDGET LINK TO PERSINFO
```

You can include information that's not in a relational database. In the following example, the EMPINFO cursor is linked to an indexed subfile, PROJECTMASTER.

```
>   DECLARE EMPINFO CURSOR FOR &
>   SELECT EMPLOYEEID, LASTNAME, &
>      PROJECTID, BILLINGAMOUNT &
>      FROM EMPLOYEES, BILLINGS &
>      WHERE EMPLOYEES.EMPLOYEEID = BILLINGS.EMPLOYEEID &
>      ORDER BY PROJECTID, EMPLOYEEID
>
>   ACCESS EMPINFO LINK TO *PROJECTMASTER
>   SORTED ON PROJECTID
```

Using Cursors in QTP

Cursors are used in the input phase of QTP much as they are used in QUIZ.

The scope of a cursor definition depends on when it is declared. Cursors that are declared:

- after the REQUEST statement may be referenced within that request.
- after the RUN statement, but before the first REQUEST statement, may be referenced anywhere within the run.
- before the RUN statement are valid for the duration of the QTP session or until a CANCEL is encountered.

As in QUIZ, the ACCESS statement in QTP may refer to one or more declared cursors (in addition to referring to subfiles or files declared in the dictionary). You can also use the LOOKUP option of the EDIT statement to refer to a declared cursor.

Parameter values for cursors can be provided in many ways, including using values from DEFINE or GLOBAL TEMPORARY items. For example:

```
>   SET LIST SQL
>   REQUEST ONE
>   DEFINE DEMPNAME CHARACTER SIZE 10 = PARM &
>      PROMPT "Enter last name: "
>   SQL DECLARE EMPINFO CURSOR FOR &
>      SELECT * FROM EMPLOYEES &
>      WHERE LAST_NAME = :DEMPNAME
>
>   ACCESS EMPINFO
__ Sql after PowerHouse variables/expression are removed:
__ SELECT *
__    FROM EMPLOYEES
__    WHERE LAST_NAME = :phE1
```

Customizing Cursors in PowerHouse

In PowerHouse, a single cursor declaration can be used in a variety of ways. It can improve productivity in application development and maintenance by reducing the number of explicit cursor declarations required in an application.

The WHERE and ORDER BY options of many cursor declarations are well-suited to customization, because often you want to retrieve the same columns, but specify the ordering or selection criteria based on other conditions. Rather than requiring a separate cursor for each possible ordering or selection requirement, PowerHouse allows cursors to be customized.

Customization can take two forms: default customization built into PowerHouse, and customization specified by the designer. In effect, you can create a "template" cursor, then provide additional specifications to customize the template for particular tasks.

To use designer-specified customizable cursors, you need to:

- indicate which part(s) of the cursor declaration are allowed to have a variety of values
- assign variable names to each of these portions so that proper substitutions can be made when the cursor is used
- specify the default options that should be used

You do this by using substitution variables on the DECLARE CURSOR statement. A substitution-variable has the general form

::name[(text)]

The double colon identifies the name as a substitution-variable and the optional text provides a default value for the substitution if no other value is provided. Any valid PowerHouse identifier can be used as the name for the variable. Prior to parsing, all substitution-variables in the DECLARE CURSOR statement are replaced with substitution values, the default text values in parentheses, or empty strings.

Default customizations are available for all cursors, including default cursors in QDESIGN. Two predefined substitution variables, ::WHERE and ::ORDERBY, exist for all cursors and allow you to augment or replace the WHERE and ORDER BY clauses used when the cursor is opened.

The values to be used for these variables can be specified using the WHERE and ORDERBY options available on statements such as the CURSOR and ACCESS statements in QDESIGN, and the ACCESS and CHOOSE statements in QUIZ and QTP. (See the next section for more information about providing substitution values.) The VIA and USING options of the ACCESS statement in QDESIGN also customize a cursor's definition, allowing multiple "access paths" to be specified for a single cursor.

For example,

```
>   DECLARE PROVINCE_DATA CURSOR FOR &
>      SELECT PROVINCE, ...FROM PROVINCES
```

```
>   CURSOR PROVINCE_DATA ...WHERE(AREA > 100)
>      ACCESS VIA PROVINCE REQUEST PROVINCE
>      ACCESS VIA CAPITAL REQUEST CAPITAL
```

PowerHouse merges these options with the initial cursor declaration as required. For more information about how customizations are merged with existing ORDER BY and WHERE clauses in cursor declarations, see (p. 28) and (p. 29), respectively.

Substituting Values

Substitution values give you the ability to customize the values within the SQL query.

Each substitution is specified as:

variable-name(text)

where variable-name corresponds to the name that was prefaced by a double colon, used in the cursor declaration and the text in parentheses is the default value. The specified text will replace the double colon, the variable name, and the optional default text in the query.

You can specify substitutions on the following statements:

QDESIGN	ACCESS, CURSOR, FIELD LOOKUP option, SQL OPEN
QUIZ	ACCESS, CHOOSE
QTP	ACCESS, CHOOSE, EDIT LOOKUP option

As a general rule, substitutions follow immediately after the name of a cursor in a statement. The exceptions are the QDESIGN ACCESS and the QUIZ and QTP CHOOSE statements, where substitutions are immediately after the first keyword of the statement.

For example, using the default values within the cursor, data is retrieved by employee number:

```
>   SET LIST SQL
>   SET DATABASE EMPBASE
>   SQL DECLARE EMPLOYEE CURSOR FOR &
>   SELECT * FROM EMPLOYEES &
>        ::WHERE &
>        ::ORDERBY(ORDER BY EMPLOYEE)
>   SCREEN EMPSCR
>   CURSOR EMPLOYEE PRIMARY KEY EMPLOYEE
.
.
.
__ Sql after PowerHouse variables/expression are removed:
__ SELECT *
__    FROM EMPLOYEES
__    ORDER BY EMPLOYEE
```

The same cursor declaration can be used for many screens. However, the use of substitutions can customize the cursor for each screen. In the following example, the same cursor is used, but data is retrieved by last name:

```
>   SET LIST SQL
>   SET DATABASE EMPBASE
>   SQL DECLARE EMPLOYEE CURSOR FOR &
>   SELECT * FROM EMPLOYEES &
>        ::WHERE &
>        ::ORDERBY(ORDER BY EMPLOYEE)
>   SCREEN NAMESCR
>   CURSOR EMPLOYEE PRIMARY KEY EMPLOYEE
>   ACCESS ORDERBY(ORDER BY LAST_NAME)
.
.
.
__ Sql after substitutions are applied:
__ SELECT *
__    FROM EMPLOYEES ::WHERE
```

```
__     ORDER BY LAST_NAME
```

An error message is issued if the substitution causes invalid syntax.

It is not necessary to use explicit variables in the DECLARE CURSOR statement to get the benefits of substitution. In a variety of cases, PowerHouse needs to change the query specified in the DECLARE CURSOR statement. For example, the ORDERBY and ORDERED options on the ACCESS statement determine the appropriate ORDER BY for the cursor. The path and the selection values (whether exact values, ranges, patterns, or generic specifications) that you, as the user provide, are incorporated into the WHERE clause for the cursor.

PowerHouse applies intelligent defaults to determine whether (and where) a WHERE option and/or ORDERBY option need to be inserted.

The following are more examples of cursor declarations and substitutions.

In this example, the OPEN statement specifies a literal text substitution.

```
>   SQL DECLARE GETAPROJECT1 CURSOR FOR &
>      SELECT * FROM PROJECTDETAILS &
>      WHERE PROJID = :PROJID &
>      ::ORDERCLAUSE
.
.
.
>   SQL OPEN getaproject1 &
>   ORDERCLAUSE(ORDER BY budgetamount DESC)
```

In QUIZ and QTP, you specify substitutions on the ACCESS statement. For example:

```
>   SQL DECLARE GETAPROJECT3 CURSOR FOR &
>      SELECT * FROM PROJECTDETAILS &
>      WHERE ::WHERECOND &
>      ::OTHERCONDITIONS &
>      ORDER BY PROJECTDATE &
>       ::SORTORDER
.
.
.
>   ACCESS getaproject3 &
>      WHERECOND(STATUS='A'), &
>      OTHERCONDITIONS(AND (budgetamount
>             BETWEEN 1000 AND 2000)), &
>             SORTORDER(ASC)
```

After substitutions, the effective cursor definition is:

```
>   SQL DECLARE GETAPROJECT3 CURSOR FOR &
>      SELECT * FROM PROJECTDETAILS &
>      WHERE STATUS= 'A' &
>       AND (budgetamount BETWEEN 1000 AND 2000) &
>      ORDER BY PROJECTDATE ASC
```

You can also nest substitutions. For example:

```
>   SQL DECLARE selectaproject CURSOR FOR  &
>      SELECT * FROM projects &
>   ::WHERE_CLAUSE &
>   ORDER BY ::ORDER
.
.
.
>   SQL OPEN selectaproject  &
>      WHERE_CLAUSE(WHERE ::ORDER < 10), &
>         ORDER(daysremaining)
```

After substitutions, the effective cursor definition is:

```
>   SQL DECLARE selectaproject CURSOR FOR &
>      SELECT * FROM projects &
>      WHERE daysremaining < 10 &
>      ORDER BY daysremaining
```

Linking Cursors

When retrieving data from multiple tables in the same database, you should use the WHERE option or the JOIN...ON option of the query specification to define how the tables are related.

```
>   SQL DECLARE EMPSKILL CURSOR FOR &
>       SELECT * FROM EMPLOYEES, SKILLS &
>       WHERE EMPLOYEES.EMPLOYNO = SKILLS.EMPLOYNO
```

A cursor can only refer to data in one database. To link multiple cursors, you can use program variables or substitutions.

Linking Using Program Variables

If the data you want to process comes from tables that are in different databases, there are a number of places where you can specify this relationship. In the following example, the link information is specified on the DECLARE CURSOR statement.

```
>   SQL IN EMPDB DECLARE EMP CURSOR FOR &
>       SELECT * FROM EMPLOYEES
>
>   SQL IN SKILLDB DECLARE EMPSKILL CURSOR FOR &
>       SELECT * FROM SKILLS &
>       WHERE EMPLOYNO = :EMPLOYNO OF EMP
>
>   ACCESS EMP LINK TO EMPSKILL
```

The text, ":EMPLOYNO OF EMP", is a program variable. A colon is used to introduce a PowerHouse variable or parenthesized expression. These program variables are used to specify values that are to be used in the SQL statement. The variables and expressions are evaluated just before the SQL statement is executed. The program variable in this case will refer to the EMPLOYNO item in the EMP file, which is the primary record structure of the ACCESS statement. The cursor EMPSKILL is re-executed for every row of data that is returned from the EMP cursor.

Linking using Substitution Variables

Instead of specifying a WHERE option in the DECLARE CURSOR statement, you can use simpler DECLARE CURSOR statements and specify the WHERE option using a substitution on the ACCESS statement. The code for this is shown in the example below. In the following example:

- Substitutions for the two default substitution variables (WHERE and ORDER BY) can be declared even if the variables do not appear on the DECLARE CURSOR statement.
- The substitution text "EMPLOYNO = :EMPLOYNO OF EMP" becomes the WHERE option of the SQL query that is passed to the database.

```
> SQL IN EMPDB DECLARE EMP CURSOR FOR &
>    SELECT * FROM EMPLOYEES
>
> SQL IN SKILLDB DECLARE EMPSKILL CURSOR FOR &
>    SELECT * FROM SKILLS
>
> ACCESS EMP LINK TO &
>        EMPSKILL WHERE (EMPLOYNO = :EMPLOYNO OF EMP)
```

Substitution variables are useful when there are complicated queries in cursors. Rather than letting every application repeat the definition of the DECLARE CURSOR, you can create a more general purpose DECLARE CURSOR statement that functions as a template. For example, assume that you want to display current and historical sales data together. The data is accessible via the EMPLOYNO column. The SALES cursor declared in the following example is a template for this query: it has a substitution variable called EMP_COLUMN, which is identified with a double colon. The default is given in parentheses, which is the program variable EMPLOYNO starting with a single colon (:).

```
> SQL IN SALESDB DECLARE SALES CURSOR FOR &
>    SELECT SALE FROM SALES &
>     WHERE EMPLOYNO = ::EMP_COLUMN (:EMPLOYNO) &
>    UNION &
>    SELECT sale FROM OLD_SALES &
```

```
>     WHERE EMPLOYNO = ::EMP_COLUMN (:EMPLOYNO)
```

This cursor can be used in the following ACCESS statement, which links salesperson information to the sales information. The ACCESS statement also specifies the text that is used for the substitution variable EMP_COLUMN.

```
> SQL IN EMPDB DECLARE SALESPERSON CURSOR FOR &
>    SELECT EMPLOYNO FROM SALESPERSON
>
> ACCESS SALESPERSON LINK TO &
>        SALES EMP_COLUMN (:EMPLOYNO OF SALESPERSON)
```

The result of the substitution is the following SQL:

```
>    SELECT SALE FROM SALES &
>      WHERE EMPLOYNO = :EMPLOYNO OF SALESPERSON &
>    UNION &
>    SELECT SALE FROM OLD_SALES &
>     WHERE EMPLOYNO = :EMPLOYNO OF SALESPERSON
```

But suppose that in most cases the historical data should be limited with a certain condition, which varies from program to program. You could enhance the template for this.

In the following example, two substitutions, EMP_COLUMN and EXTRA_CONDITION, are specified on the ACCESS statement.

```
> SQL IN SALEDB DECLARE SALES CURSOR FOR &
>    SELECT SALE FROM SALES &
>      WHERE EMPLOYNO = ::EMP_COLUMN (:EMPLOYNO) &
>   UNION &
>    SELECT SALE FROM OLD_SALES &
>      WHERE EMPLOYNO = ::EMP_COLUMN (:EMPLOYNO)  &
>     ::EXTRA_CONDITION
>
> ACCESS SALESPERSON LINK TO &
>    SALES EMP_COLUMN (:EMPLOYNO OF SALESPERSON), &
>          EXTRA_CONDITION (AND YEAR >= 2001)
```

The resultant SQL query for the SALES cursor is:

```
>  SELECT SALE FROM SALES &
>         WHERE EMPLOYNO = :EMPLOYNO OF SALESPERSON &
>      UNION &
>  SELECT SALE FROM OLD_SALES &
>         WHERE EMPLOYNO = :EMPLOYNO OF SALESPERSON &
>         AND YEAR >= 2001
```

There are some additional rules regarding substitutions that address the customization that you can use when you want PowerHouse to merge WHERE conditions and ORDER BY criteria. These rules are outlined at the end of this section.

SQL Reserved Words

The following reserved words cannot be used as database object names in SQL in PowerHouse. To use a reserved word as a database object name (table name, column name, and so on), it must be enclosed in double quotes.

For example:

```
>   SELECT "SELECT" FROM "FROM"
```

SQL Reserved Words

all	escape	month	snapshot
and	exists	multidatabase	some
any	extract	natural	substring
as	filename	minute	set

SQL Reserved Words			
asc	first	not	sum
at	fn	null	table
avg	for	octet_length	temporary
between	from	of	then
by	group	oj	union
call	having	on	update
case	hour	or	upper
char_length	in	order	user
count	inner	out	values
create	insert	outer	when
cross	into	position	where
current	is	ravg	xavg
day	join	rcount	xcount
dbkey	keep	returning	xfirst
delete	last	right	xlast
desc	left	rmax	xmax
distinct	like	rmin	xmin
drop	lower	rsum	xsum
else	max	second	year
end	min	select	

Substitution Rules for ORDERBY

PowerHouse needs to insert WHERE conditions and ORDER BY columns into the query of a DECLARE CURSOR. The following is a list of substitution rules, in order of precedence, that PowerHouse uses to insert ORDER BY columns into a query:

1. If ::ORDERBY exists, it is replaced with the ORDER BY keyword and the list of column names following the ORDERBY variable.
2. If ::COMMA_ORDERBY exists, it is replaced with ", order-column-list".

   Note: order-column-list represents a list of columns you specify for ordering.
3. If ::ORDERBY_COMMA exists, it is replaced with "order-column-list ,".
4. PowerHouse inserts ::COMMA_ORDERBY after the list of existing columns on the first ORDER keyword. This will be substituted with ", order-column-list".
5. PowerHouse appends an ::ORDERBY to the statement. This is substituted as in Rule 1.

In the following example, PowerHouse adds an ORDER BY option to the query using Rule 5:

```
>   DECLARE GETAPROJECT1 CURSOR FOR &
>       SELECT * FROM PROJECTDETAILS
```

```
>   CURSOR GETAPROJECT1 PRIMARY KEY PROJECT
>      ACCESS VIA PROJID ORDERED REQUEST PROJID
```

The resultant SQL would be:

```
SELECT * FROM PROJECTDETAILS
WHERE PROJID=:PROJID
ORDER BY PROJID
```

These rules also apply in more general cases. When a cursor is used, you can specify an ORDERBY(order-column-list) substitution which will be inserted in the query. The ORDERBY variable is a predefined substitution variable.

Substitution Rules for WHERE

The following is a list of substitution rules, in order of precedence, that PowerHouse uses to insert a WHERE condition into an SQL query:

1. If ::WHERE_CLAUSE exists, it is replaced with "WHERE where-text"

   Note: where-text represents the value of the WHERE option. Note that the default value specified for the substitution should not begin with WHERE.
2. If ::AND_CONDITION exists, it is replaced with"AND where-text"
3. If ::CONDITION_AND exists, it is replaced with "where-text AND"
4. PowerHouse inserts ::CONDITION_AND after the first WHERE keyword. This will be replaced with "where-text AND".
5. PowerHouse inserts ::WHERE_CLAUSE before the first ORDER/GROUP/HAVING/UNION keyword.
6. PowerHouse appends ::WHERE_CLAUSE to the statement and substitute as under 1.

WHERE is a predefined substitution variable. You can specify a WHERE(where-text) substitution which will be inserted in the query. In the following example, the specified condition is inserted into the query of the DECLARE CURSOR PRICE using Rule 6.

```
>   DECLARE PART CURSOR FOR SELECT * FROM PARTS
>   DECLARE PRICE CURSOR FOR SELECT * FROM PRICE
>   ACCESS PART LINK TO PRICE &
>      WHERE(PRICE.ID=:PRICEID OF PART)
```

Developer-Written SQL Queries

There is a potential issue regarding developer-written SQL queries affecting PowerHouse Series 8 in that all data records may not be retrieved under certain specific circumstances during distributed sorting.

Distributed sorting means that one or more physical queries are sorted by the database using its collating sequence, and one or more physical queries are sorted using the operating system collating sequence on the computer running the Cognos product, or are sorted by another database using its collating sequence. You may have the problem with distributed sorting if

- the collating sequences or sort orders are different,

  and
- the SQL query is written by the application developer and inserted into the PowerHouse 4GL code,

  and
- the SQL query written by the application developer and inserted into the PowerHouse 4GL code is sufficiently complex to be decomposed into two or more physical database queries by the Cognos Services Layer. The two or more physical queries are subsequently merged together after distributed sorting. An example of a complex logical query is a query that has groupings with subtotals containing extended aggregates, such as moving averages or rolling subtotals.

You will not have the problem with distributed sorting if

- the data is sorted in the same order by all your databases and by your operating system.

- there is no SQL query written by the application developer and inserted into the PowerHouse 4GL code because PowerHouse 4GL, PowerHouse Web, and Axiant do not generate sufficiently complex SQL on their own to cause distributed sort problems.

An enhancement has been added to PowerHouse that will allow you to ensure that distributed sorts are not used when processing multiple SQL statements using the database connection. If you insert SQL queries into your PowerHouse code and you use different collating sequences, you should use the following functionality to avoid data integrity problems.

In PDL, the DATABASE OPEN name string may be appended with the following string:

```
@COLSEQ=NOT_COMPATIBLE
```

This value will ensure that distributed sorts are not used when processing multiple SQL statements using this database connection.

The @COLSEQ=NOT_COMPATIBLE string must be the last part of the open name string. If the DATABASE statement has associated USERID and PASSWORD options, the @COLSEQ=NOT_COMPATIBLE string may be placed at the end of the PASSWORD value in order to ensure it is appended to the end of the open name string.

Depending on the behavior your database supports

- the ORDER BY clauses of all SQL statements will be sent for database execution, providing maximized performance
- no ORDER BY clause is sent in any SQL statement. In this case there is a potential for some performance degradation, because the ORDER BY clauses are executed on the platform running your PowerHouse, PowerHouse Web, or Axiant product. The amount of degradation varies by query and database so no reliable estimate is possible.

Stored Procedures: RDBMS Specifics

Stored procedures and stored functions are collections of SQL statements and logic that are stored in a database. Stored procedure calls are available from the DECLARE CURSOR (stored procedure) statement, the SQL CALL verb in QDESIGN, and the SQL CALL statement in QTP. For general information on this syntax, see the *QDESIGN Reference, QUIZ Reference,* and *QTP Reference* books.

Issues that are specific to each RDBMS are identified below.

Oracle Stored Procedures

Syntax for Stored Procedure Names

The syntax for an Oracle procedure name is:

[owner-name.[package-name.]]procedure-name|
function-name[@database-linkname]

If the package-name or database-linkname is included, they are treated as part of the procedure or function name, and double quotes are required. For example,

```
MANAGER."MTHEND_PROCEDURE@DBLNK01"
MANAGER."MTHENDPKG.MTHEND_PROCEDURE"
```

Oracle synonyms may be used for package-names, procedure-names, and function-names. For more information about how PowerHouse uses Oracle synonyms, see Chapter 5, "PowerHouse Language Rules", in the *PowerHouse Rules* book.

Result Sets

PowerHouse supports Oracle stored procedures returning result sets using a variable with a datatype of REF CURSOR. The REF CURSOR variable is used to pass query result sets between PL/SQL stored subprograms and various clients such as PowerHouse. Use the OPEN statement with a FOR to open the variable. The syntax is:

```
OPEN { cursor_variable_name } FOR select_statement;
```

For example, to create stored procedures that return result sets, use the following SQL statement in Oracle SQL Plus to create the variable:

```
CREATE PACKAGE MYPKG
IS
  TYPE CursorType IS REF CURSOR;
END MyPkg;
```

Then use the variable in the stored procedure as in:

```
CREATE PROCEDURE RETURN_RESULT_SET
 (oCursor IN OUT MyPkg.CursorType) AS
BEGIN
  open oCursor for select * from TableName;
END;
```

Oracle stored procedures that return result sets are called from PowerHouse in the same way as other database stored procedures. When using the Result Set syntax, the item portion cannot be a cursor variable. It must be the column in the table. For example, if the OPEN in the stored procedure looks like this:

```
open oCursor for select employee, city from employees;
```

The syntax in PowerHouse should be:

```
> DECLARE Test_Return CURSOR FOR CALL RETURN_RESULT_SET
...
>   RESULT SET employee num(4), city char(20)
```

PowerHouse does not support the JDBC or SQLJ drivers with Oracle.

Sybase Stored Procedures

The syntax for a Sybase procedure name is:

[server-name.][database-name.][owner-name.]procedure-name

If server-name is included in a Sybase procedure name, double quotes are required for the server-name and database-name. For example,

```
"DBSVR01.ACCNT".MANAGER.MNTHEND_PROCEDURE
```

DB2 Stored Procedures

PowerHouse supports stored procedures for DB2. PowerHouse supports

- output parameters
- input parameters
- INOUT parameters
- result sets

Stored procedures can be written using SQL, Java, C, or C++.

A stored procedure can be called using SQL CALL or by running an executable that is used to call the stored procedure.

The IBM DB2 Stored Procedure Builder (SPB) can be used to help develop stored procedures.

Syntax for Stored Procedure Names

The syntax for a DB2 procedure name is:

[owner-name.]procedure-name

Result Sets

To return a result from a stored procedure, you must

1. Declare a cursor on that result set.
2. Open a cursor on that result set.
3. Leave the cursor open when exiting the procedure.

If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened.

The following is an example of DB2 SQL procedure definition with a result set:

```
    CREATE PROCEDURE ADT.slct_mngr_rs ()
    LANGUAGE SQL
    DYNAMIC RESULT SETS 1
BEGIN
   DECLARE cur1 CURSOR WITH RETURN TO CALLER FOR
   SELECT BRANCH_MANAGER from adt.branches;
   OPEN cur1;
END
```

To call this stored procedure from PowerHouse QTP, you can use one of the following examples of [SQL] CALL syntax:

```
sql in <databasename> on errors report call adt.slct_mngr_rs
```

or

```
sql declare ccminfo cursor for &
call adt.slct_mngr_rs result set branch_manager varchar(20)
ACCESS ccminfo
```

Size Matching in Java Stored Procedures

To prevent possible errors when developing Java stored procedures, it is important to ensure that item sizes match.

If you define a decimal in Java, for example, dec(3,0), "3" is a precision value in Java. To define a global temporary item size in QTP to match what is defined in Java, the size of the integer should be calculated as:

```
precision = length * 2  - 1
```

The global temporary in QTP would be defined as:

```
global temporary bcount integer size 2
```

ODBC (including Microsoft SQL Server) Stored Procedures

Stored procedure calls are supported for ODBC but depend on the capabilities of the data source. Calls to stored procedures can take input parameters from a calling program, return values for output parameters to a calling program, and return status values indicating success or failure. A stored procedure may also return result sets.

The returning item identifies the item that contains the return status from a stored procedure upon completion of the stored procedure. It must be a 32-bit integer.

Syntax for Stored Procedure Names

The syntax for an ODBC procedure name is:

[server-name.][database-name.][owner-name.]procedure-name

Result Sets

In QUIZ, QTP, and QDESIGN, you can write a DECLARE CURSOR containing a CALL statement with a result set. For example:

```
DECLARE X CURSOR FOR &
   CALL Y &
   RESULT SET &
   EMPLOYEE_NO SMALLINT, &
   LAST_NAME CHAR(15), &
   BILLING QUADWORD
```

You must ensure that the datatype in the result set matches the datatype in the relational database. If you have a Microsoft SQL Server column with datatype 'MONEY', the PowerHouse result set datatype must be 'MONEY'. One exception is the Sybase datatype ’MONEY’, which maps to the PowerHouse result set datatype ’QUADWORD’ due to internal storage differences.

Oracle Rdb Stored Procedures

Oracle Rdb supports stored procedures but not stored functions. The database must be declared as TYPE RDB in the dictionary. Oracle Rdb databases declared as TYPE RDB/VMS do not support direct usage of SQL including stored procedures.

Syntax for Stored Procedure Names

The syntax for an Oracle Rdb procedure name is:

procedure-name

Creating User-Defined Functions (DB2, Oracle)

Application administrators may wish to provide their PowerHouse developers with access to functions beyond those already provided by PowerHouse. PowerHouse provides the ability to call outside functions from within its components. These can be either defined in the database or in external libraries (DLLs on Windows or shared libraries on UNIX). They are referred to as user-defined functions (UDFs), where 'user' represents the administrator who has created the function and placed it in the database or library. 'User' does not represent the screen, run, or report user, who simply invoke the function once it has been defined.

Refer to your Oracle or DB2 documentation for information on creating UDFs.

For both database functions and external functions, PowerHouse supports only scalar functions. A scalar function returns a single value each time it is invoked. Aggregate functions (those returning a set of data) are not supported.

Calling UDFs from PowerHouse

UDFs can be called from within PowerHouse in two ways:

- Oracle UDFs (also known as stored functions) can be called by the QDESIGN CALL verb and the QTP CALL statement, without any PowerHouse preparation.
- Both Oracle and DB2 UDFs can be called as part of an SQL expression in QDESIGN, QTP, and QUIZ. However, you must first create and edit a database-specific SQL file that declares the UDF properties to PowerHouse. Examples of SQL expressions are:
  - the select list of a SELECT statement
  - the condition of a WHERE clause
  - ORDER BY, and GROUP BY clauses

Creating the Database-Specific File: cogudfor.sql and cogudfd2.sql

For PowerHouse to support database and external UDFs, you must declare the properties of the UDF in the database-specific file, **cogudfor.sql** (**Oracle**) or **cogudfd2.sql** (**DB2**). To create this file, copy the template file cogudf.sql provided in the PowerHouse install directory to either cogudfor.sql or cogudfd2.sql.

Database-Specific Files in the Windows Environment

In Windows, the database-specific file must be located in one of the following directories:

- the current working directory
- the directory which is denoted by the value of the UDF SQL Directory in the appropriate [Services] section of the axiant.ini or powerhouse.ini file
- the directory where the Cognos product is located

Database-Specific Files in the UNIX Environment

For UNIX, the database-specific file must be located in one (and only one) of the following locations:

- the current working directory

- the environment variable COGUDFSQL
- the environment variable DMDBINI

Examples of the environment variables are:

```
COGUDFSQL = <path to directory with cogudfd2.sql>
export COGUDFSQL
DMDBINI = <path to directory with cogudfd2sql>
export DMDBINI
```

Declaring the UDF Properties in the Database-Specific File

The database-specific file cannot contain any Cognos SQL data manipulation language (DML) statements, such as SELECT or INSERT. Use the following syntax to declare the properties of a UDF to PowerHouse. This syntax applies to both database and external user-defined functions (p. 36).

```
<database function declaration> ::=
  DECLARE [ DATABASE] [ <function type> ] FUNCTION
  <function name> <formal parameter list>
  RETURNS <data type>
  FUNCTION NAME <database function name> ;

<function type> ::=
      SCALAR | AGGREGATE

<function name> ::=
      [<local name>.]<identifier>

<formal parameter list> ::=
      [ ( [ <data type> [ { , <data type> } … ] ] ) ]

<returns data type> ::=
      <data type>
<database function name> ::=
    : [<catalog>.][<schema>.]<function name>

<local name> ::=
  : <identifier>

<catalog> ::=
  : <identifier>

<schema> ::=
  : <identifier>

<function name> ::=
  : <identifier>

<identifier> ::=
  : text
  | "<text>"

<data type> ::=
      STRING
    | BOOLEAN
    | NUMBER
    | BINARY
    | DATE
    | TIME
    | TIMESTAMP
    | INTERVAL
    | BLOB
    | TEXT
    | <literal value>

<literal value> ::=
  : '<text>'
```

Example (Oracle)

Adding the UDF to the Database

The following simple example returns the salary with 10% tax added.

1) Create a table.

```
CREATE TABLE Tax_table (
Ss_no NUMBER,
Sal NUMBER);
```

2) Create a user-defined function

```
CREATE OR REPLACE FUNCTION tax_rate (ssn IN NUMBER, salary IN
NUMBER) RETURN NUMBER IS
sal_out NUMBER;
BEGIN
sal_out := salary * 1.1;
return (sal_out);
END;
```

Declaring the UDF in the Database-Specific SQL File

The following UDF declaration must be added to the database-specific SQL file, cogudfor.sql:

```
DECLARE FUNCTION tax_rate(NUMBER, NUMBER)
RETURNS NUMBER
FUNCTION NAME scott.tax_rate;
```

Calling the UDF from PowerHouse

To call this function from QDESIGN, QUIZ, or QTP, you use the following syntax:

```
SQL DECLARE cursor_1 CURSOR FOR &
    SELECT tax_rate(Ss_no, Sal) FROM Tax_table
```

Oracle functions may also be called from a QDESIGN CALL verb or a QTP CALL statement without having to declare it in cogudfor.sql. For example:

```
SQL call tax_rate (x in, y in) returning z
```

Example (DB2)

Adding the UDF to the Database

The following simple example returns the last name of employee number 5.

First create a table and an index:

```
CREATE TABLE EMPLOYEES (                                    \
     EMPLOYEE_NO                          INTEGER not null, \
     FIRST_NAME                           VARCHAR(12),      \
     LAST_NAME                            VARCHAR(20),      \
     PHONE                                CHAR(10),         \
     POSITION                             CHAR(2),          \
     DATE_JOINED                          TIMESTAMP)

CREATE UNIQUE INDEX EMPLOYEES_EMPLOYEE  \
        on  EMPLOYEES(EMPLOYEE_NO)
```

Then create a user-defined function:

```
CREATE FUNCTION selectemp ()
RETURNS VARCHAR(20)
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
DETERMINISTIC
RETURN
SELECT LAST_NAME
FROM EMPLOYEES
WHERE EMPLOYEE_NO = 5
```

Declaring the UDF in the Database-Specific SQL File

The following UDF declaration must be added to the database-specific SQL file, cogudfd2.sql:

```
DECLARE FUNCTION substitute(STRING, STRING)
RETURNS STRING
FUNCTION NAME scott.tiger.substitute;
```

Calling the UDF from PowerHouse

To call this function from QDESIGN, QUIZ, or QTP, you use the following syntax:

```
SQL DECLARE ccminfo CURSOR FOR &
    SELECT employee FROM employees WHERE last_name = selectemp ()
```

External User-Defined Function (DB2, Oracle) and External Procedure (Oracle) Support

In addition to calling user-defined functions that have been defined in the database, you can also call UDFs that have been compiled into external libraries (DLLs on Windows or shared libraries on UNIX). Oracle refers to these as External Procedures or External Routines. To successfully create external UDFs for use in PowerHouse, you must

- satisfy certain UDF requirements
- use the designated files provided by PowerHouse
- create external UDF libraries
- register the UDF to the RDBMS
- create and edit the database-specific file
- call the UDF

UDF Requirements and Restrictions

Here is a list of certain requirements and restrictions that you must satisfy when creating external UDFs for PowerHouse.

- **DB2**: You can use a programming language other than C to build external UDFs, but the UDFs must support the C calling convention.
- The name of the external UDF must be unique.
- The maximum number of parameters is 16.
- Parameters cannot assume more than one data type.
- Binary, text, and BLOBs are not supported as either parameters to, or return values from, external UDFs.
- All data items supplied to and returned by an external UDF are aligned, dependent upon the data types.
- **DB2**: All external UDFs must return a void value.)

Use Designated Files

Cogudf.h

Cogudf.h defines all macros, types, and functions required to interface with the UDF capability. Oracle UDFs can use this file or they can use their own header file.

Cogudfty.h

Cogudfty.h defines the data types supported in user-defined functions. It is included by the cogudf.h. Oracle UDFs can use this file or they can use their own header file.

Create External UDF Libraries

DB2: You can use the C or C++ programing languages to build external UDF libraries. If languages other than these are used, they must support the C language calling convention. In addition, any DLL used to support external UDFs must be a 32-bit DLL.

Oracle: An external procedure is stored in a dynamic link library (DLL), or libunit in the case of a Java class method.

The cogudf.h file provides access to all the necessary macros, type definitions, and function definitions. The cogudf.h file includes cogudfty.h, so you do not need to explicitly include the latter.

Windows Environment (DB2)

In a Windows environment, all DB2 UDFs must be exported from the DLL in which they reside to be accessible to your Cognos product. Your Cognos product can simultaneously access one or more DLLs with UDFs.

Here is a list of additional requirements to remember when exporting UDFs:

- All DLLs must be 32 bit and compiled with 8-byte alignment.
- DLLs containing UDFs must be located in one of the following locations:
  - the current directory of the application to be executed
  - one of the directories specified in the PATH environment variable
  - the Windows directory
  - the Windows system directory
- To export your UDFs, you must create an export file (.def). For more information, see your DB2 documentation.

UNIX Environment (DB2)

In a UNIX environment, DB2 UDFs may exist in any number of shared libraries. The UDF shared library must be located in a directory accessible to the run-time linker. You can point to a UDF shared library using the UNIX environment variables

- LD_LIBRARY_PATH (Solaris)
- LIBPATH (AIX)
- SHLIB_PATH (HP-UX)

Register Your User-Defined Functions and External Procedures

To use your UDFs, you must register them in Oracle or DB2.

Oracle: In SQL*Plus, use the CREATE FUNCTION call as in the following example:

```
CREATE FUNCTION find_max(
x IN BINARY_INTEGER,
y IN BINARY_INTEGER)
RETURN BINARY_INTEGER AS
   EXTERNAL LIBRARY udflib
   NAME "find_max"
   LANGUAGE C;
```

You can then use the CREATE PROCEDURE call as in the following example:

```
CREATE OR REPLACE PROCEDURE UseIt AS
    a integer;
    b integer;
    c integer;
BEGIN
    a:=1;
    b:=2;
    c:=find_max(a,b);
END;
```

DB2: In the Command Line Processor (CLP) window in DB2, you can enter the SQL statements to register your UDFs. The function name you register in your database must match the name used in your external programming.

In the following example, the function name ScalarUDF in the library udfsrv, is registered in the DB2 database. This name is used to link to the library.

```
CREATE FUNCTION ScalarUDF (VARCHAR(20), CHAR(2)) RETURNS char
FENCED
EXTERNAL NAME 'udfsrv!ScalarUDF'
```

```
NOT VARIANT NO SQL PARAMETER STYLE DB2SQL LANGUAGE C
NO EXTERNAL ACTION
```

Add Function Definitions to the Database-Specific File

After you have created your function definitions, you must declare them in the PowerHouse database-specific file. For information about how to create and edit the file, see (p. 33) and (p. 34).

Here is an example of a database UDF declaration in cogudfd2.sql:

```
DECLARE DATABASE FUNCTION ScalarUDF(string,string)
RETURNS STRING
FUNCTION NAME adt.ScalarUDF;
```

This is an example of adding find_max into cogudfor.sql:

```
DECLARE DATABASE FUNCTION find_max(number,number)
RETURNS number
FUNCTION NAME scott.find_max;
```

Call External UDFs

After you have defined and registered your external UDF, you can now call the UDF from PowerHouse. This is done in the same manner as for database UDFs, as described on (p. 33).

Tracing UDF File Errors

Any errors that occur while parsing a function definition file are written to a UDF log file. No syntax or semantic errors are reported to the user - the log file must be examined to determine what errors occurred, if any. The UDF log file is intended as an aid to IS or OEM developers that are defining UDFs for use within a Cognos product.

In Windows, the log file is denoted by the value of the UDF Log File entry in the appropriate [Services] section of the axiant.ini or powerhouse.ini file. Different versions of Services applications have differently named sections.

If this entry does not exist or does not have a value, no information is written to a log file. If the file named already exists, any errors reported are simply appended to the existing file. If the file does not exist, it is created. The file is created in the directory in which the Cognos product is located, unless a path is supplied as part of the file name. An example of such an entry is:

```
[Services]
UDF Log File=d:\temp\udf.log
```

For UNIX, the value of the environment variable COGUDFLOG represents the name of the file to which errors are logged. If the environment variable does not exist, or it has no value, then no error information is written to a log file. The file is created in the current working directory, unless a path is supplied as part of the file name.

Chapter 2: Relational Support in QDESIGN

Overview

This chapter provides an overview of PowerHouse support for relational databases that are identified in your dictionary. You'll find information about

- QUICK transaction models
- overriding the transaction defaults in QUICK
- attaches and transactions in QUICK
- tuning attaches in PowerHouse
- transaction error handling in QUICK

QUICK Transaction Model Overview

PowerHouse provides a default set of high-level transaction models that make it easier to code your application. With these models, you need not specify all the transactions and transaction control required for every PowerHouse application; PowerHouse establishes default transaction attributes and timing, associates activities with transactions, and controls transaction commits and rollback. In addition, the PowerHouse processing models incorporate built-in checking for such things as update conflicts, optimistic locking, and error recovery.

When the defaults aren't sufficient, PowerHouse provides a "matrix" of options and alternatives that allow you to customize, augment, or even replace the default processing at whatever level necessary, without giving up built-in support. The sections, "Default Transaction Timing in Quick" on (p. 55), and "Overriding the Transaction Defaults in QUICK" on (p. 59), list the options available for overriding PowerHouse's default processing.

QUICK Processing Environment

The QUICK transaction models are:

- Concurrency
- Optimistic
- Consistency
- Dual

Note: The use of terms such as Concurrency, Consistency, and Optimistic in the PowerHouse context should not be confused with database product options having the same name.

Since QUICK is primarily used for multiple-user access where high data concurrency is required, the default transaction model is Concurrency. A variation on this model is called the Optimistic model.

In comparison with the Optimistic model, the Concurrency model provides more accurate checking of referential integrity at the application level, particularly in environments where reference tables are not static.

The Consistency model is often used for specialized screens (such as those performing short transactions, or requiring protection of data prior to update). This simple transaction model relies on the database to enforce data consistency. As a result, concurrency is reduced, and some of the flexibility typical of PowerHouse screens is lost.

In comparison to the Consistency model, the Concurrency model separates retrievals and updates into logically distinct activities allowing updates to be committed without terminating the retrieval chain.

In addition, PowerHouse supports a transaction model called Dual. This model allows you to design a screen to use one transaction model for Entry and Find operations, and a different model for Select operations.

Setting the Default Model

The default model and the default definition of the Dual model can be changed in the dictionary using the SYSTEM OPTIONS statement in PDL.

For more information about overriding the default model, see the SYSTEM OPTIONS statement in Chapter 2, "PDL Statements", in the *PDL and Utilities Reference* book.

For more information about overriding the default definition of the Dual model, see (p. 49).

The Concurrency Model in QUICK

The Concurrency model provides multi-user access to data and full functionality, yet still enforces a fairly high level of consistency. In this model, multiple users can read data, but only one user can update the same record at a time. When a user updates, PowerHouse verifies that there is no conflict among concurrent updates by re-reading the record and comparing the checksum of the record to the checksum of the original record. The record is updated if the checksums are equal. This approach generally results in high concurrency, since data is not locked until a user updates. This model is suitable for applications in which there are few "natural" update conflicts between users, or applications that mix data from relational and non-relational sources.

The checksum calculation omits

- calculated columns. If they were included, the values could have been changed by the database, resulting in a checksum mismatch. This can easily occur if the user does an Update Stay. Removing calculated columns from the checksum calculation eliminates these false errors.
- columns referenced by an ITEM statement with the OMIT option. The OMIT option specifically tells QUICK to exclude the column any updates typically because it is a read-only column or a calculated column. These columns are also excluded from the checksum.
- blob columns. These are excluded from the checksum calculation for performance reasons, as they can be very large.
- relational columns not referenced by the screen. These are excluded because the checksum is based on the underlying SQL generated for the QUICK screen.

Predefined Transactions

In the Concurrency model, PowerHouse defines two predefined transactions:

- Query transaction
- Update transaction

A complete chart showing all of the transaction models and associated predefined transactions appears on (p. 49).

Screen Phases

PowerHouse also divides the activities associated with a screen into three screen phases. Screen phase definitions make it easier to specify which PowerHouse transaction should be used for a specific screen activity. The screen phases are:

- Query phase
- Process phase
- Update phase

The Query transaction is a read-only transaction used for the Query phase. The Update transaction is a read/write transaction used for both the Process and Update phases.

	Query phase	Process Phase	Update Phase
Query Transaction	read-only		
Update Transaction		read/write	read/write

Query phase

QUICK uses the Query phase to determine an access path, to retrieve records for processing, and to display data. This phase encompasses the following:

Procedures:	DETAIL FIND FIND PATH POSTPATH SELECT
Procedures (in FIND and SELECT processing):	INITIALIZE
Field-level procedures that may be invoked in this phase:	INPUT OUTPUT

Field-level procedures can be invoked by the REQUEST verb and field displays. Automatic retrieval of reference files may also be included during this phase when they are invoked by the display step of the FIND mode processing cycle.

Process phase

QUICK uses the Process phase to enter or correct new records, or change existing records. Any field validation, including LOOKUP validation, is done in this phase. This phase encompasses the following:

Processing Modes:	CHANGEMODE CORRECTMODE ENTRYMODE
Procedures:	APPEND BACKOUT DESIGNER (named) DETAIL POSTFIND ENTRY EXIT MODIFY POSTFIND POSTSCROLL PREENTRY PRESCROLL
Procedure in ENTRY and APPEND processing:	INITIALIZE
Field-level procedures that may be invoked in this phase:	DESIGNER (numbered) EDIT INPUT OUTPUT PROCESS

Update phase

QUICK uses the Update phase to cover activities associated with the update procedures. This phase encompasses the following:

Procedures:	PREUPDATE UPDATE POSTUPDATE

For all screen phases, INTERNAL procedures use the phase of the calling procedure.

Screen Phases for Entry/Append Modes and Find/Select Processing

The following tables show the screen phases that correspond to the standard QUICK screen cycle for Entry/Append mode and Find/Select processing:

Screen Modes and Phases

Mode	Activity	Phase
Entry/Append	Initialization Phase	Process
Processing	Entry Sequence	Process
	Correction Phase	Process
	Update Phase	Update
Find/Select	Initialization Phase	Query
Processing	Path Determination Phase	Query
	Retrieval cycle:	
	Retrieval Initialization	Query
	Data Retrieval	Query, Process
	Display Data	Query
	Change	Process
	Update	Update

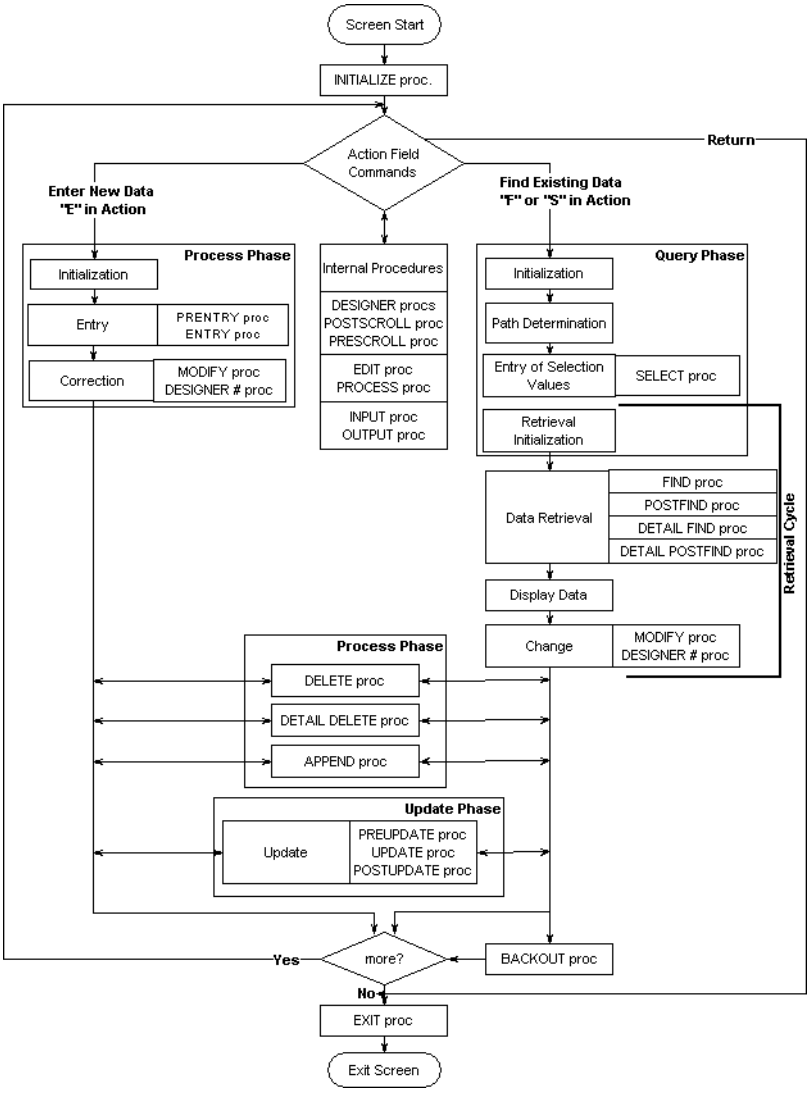
Procedures and Screen Phases

Procedure	Phase
APPEND	Process
BACKOUT	Process
DELETE	Process
DESIGNER (named)	Process
DETAIL DELETE	Process
DETAIL FIND	Query
DETAIL POSTFIND	Process

Procedures and Screen Phases (Continued)	
Procedure	**Phase**
EDIT[3]	Process
ENTRY	Process
EXIT	Process
FIND	Query
INITIALIZE[2]	Query, Process
INPUT[1,3]	Query, Process
INTERNAL PROCEDURES[3]	Process
MODIFY	Process
DESIGNER (numbered)	Process
OUTPUT[1,3]	Query, Process
PATH	Query
POSTFIND	Process
POSTPATH	Query
POSTSCROLL[3]	Process
POSTUPDATE	Update
PREENTRY	Process
PRESCROLL[3]	Process
PREUPDATE	Update
PROCESS[3]	Process
SELECT	Query
UPDATE	Update

[1] May be invoked in the QUERY phase by REQUEST verbs and field displays.
[2] When no mode is specified, initialization is in the PROCESS phase.
[3] The procedures may be in different phases depending on how they are invoked.

The following diagram illustrates the Procedural Flow and Screen Phases:



Concurrency Model Example

The following screen consists of a primary file, EMPLOYEES, and a reference file, BRANCHES, both from the same database, LIFE. The FIELD statements for EMPLOYEE_NO and BRANCH_CODE both include LOOKUP options that reference database files. Notice also that an item, BRANCH_NAME, from the reference file is displayed on the screen.

```
> SCREEN EX1 TRANSACTION MODEL CONCURRENCY
> FILE EMPLOYEES IN LIFE
> FILE BRANCHES IN LIFE REFERENCE
>
> FIELD EMPLOYEE_NO OF EMPLOYEES... &
>   LOOKUP NOTON EMPLOYEES
> FIELD LAST_NAME OF EMPLOYEES
.
.
.
> FIELD BRANCH_CODE OF EMPLOYEES... &
>   LOOKUP ON BRANCHES
> FIELD BRANCH_NAME OF BRANCHES DISPLAY
.
.
.
```

This example is used in the next section, "Concurrency Model for Oracle Rdb", and in the section, "Concurrency Model for ALLBASE/SQL, DB2, ODBC, Oracle, and Sybase", on (p. 46).

Concurrency Model for Oracle Rdb

Note: For information about the Concurrency model for ALLBASE/SQL, DB2, ODBC, Oracle, and Sybase, see (p. 46).

When you use a screen to retrieve and change existing data, you pass through all three screen phases and use both default transactions. The Query transaction retrieves the existing data, and the Update transaction performs any database updates. Several transactions can be active at the same time, including both Query and Update transactions.

When new data is entered, only the Process and Update phases are involved, and all activity takes place within the Update transaction. The Query transaction is not active.

The Update transaction starts when you retrieve or update data from a relational table during the Process phase (for example, to satisfy lookups during field processing), or send the new record to the database when the Update phase is initiated (if no earlier retrievals or updates were required). The Update transaction is committed, by default, at the end of the UPDATE procedure.

By default, QUICK does not reserve any tables for the PowerHouse transactions.

Finding Existing Data

To find or select existing records, the Query transaction retrieves the EMPLOYEES record and the BRANCHES record (to display the BRANCH_NAME information). The Query transaction is a read-only transaction, that is, it will read from the snapshot file if necessary. Snapshotting should be enabled (immediate) in Oracle Rdb, otherwise the Query transaction cannot read from the snapshot file and conflict will result.

Changing Existing Data

Recall that changes are made in the Process phase, while updates are done in the Update phase. The duration of the Update transaction, here, is from the start of the first lookup to the end of the update. This group of activities is treated as a unit for consistency and recovery purposes, and any locks acquired are held until the unit ends.

In contrast to the situation previously described, the duration of the Update transaction here is just from the start to the end of the update.

The Update transaction starts as soon as access to the database is necessary.

If you change the BRANCH_CODE field, database access is required because of the LOOKUP option on the BRANCHES relation. The Update transaction starts to perform the lookup, remains active to perform other lookups as other fields are changed, and also sends the changes to the database when you issue an UPDATE command. When the UPDATE is complete, the Update transaction is committed.

If you change only fields that do not have lookups, such as LAST_NAME, no database access is necessary. In this case, the Update transaction starts when you issue an UPDATE command, at which point the changes are sent to the database. As before, the Update transaction is committed only when the UPDATE is complete. As part of the update logic, the records to be updated are first re-retrieved and checksummed to ensure that they have not been changed.

Entering New Data

By default, the Update transaction is used for both the Process and Update phases. In the Process phase, you enter and correct data. In the Update phase, you send the new record to the database.

As previously described, the Update transaction starts as soon as it's needed. The LOOKUP option on the EMPLOYEE_NO FIELD statement causes the Update transaction to start. This Update transaction is also used for the LOOKUP option on the BRANCH_CODE field.

In Oracle Rdb, a lock may be held from the time a lookup is executed in Entry mode until the time the Update transaction is committed. This could cause a reduction in concurrency for the affected table. There are several ways to improve concurrency:

- use the AUTOCOMMIT option on the QDESIGN FILE or FIELD statement

- associate the LOOKUP with a different transaction
- use the Query transaction for the process phase of the lookup file
- use the Optimistic model

Note: A transaction doesn't have to be active to use a QUICK screen; transactions are used only to retrieve or send data to a database. Screens that do not use a LOOKUP option start an Update transaction only when the user updates, and they commit it at the end of the Update procedure.

Cursor Retention

By default, in the Concurrency model, PowerHouse retains the read chain after a commit.

Concurrency Model for ALLBASE/SQL, DB2, ODBC, Oracle, and Sybase

The Concurrency model for ALLBASE/SQL, DB2, ODBC, Oracle, and Sybase defines both the Query and Update transactions, however, the Query transaction is not used. The Update transaction is used for all screen phases; all database activities are associated with a single Update transaction.

The Update transaction starts as soon as access to the database is required, and ends when data is committed at the end of the UPDATE procedure.

Note: For information on the Concurrency model for Oracle Rdb, see (p. 45).

Finding Existing Data

To find or select existing records, the Update transaction retrieves the EMPLOYEES record and the BRANCHES record (to display the BRANCH_NAME information).

Changing Existing Data

When you use a screen to retrieve and change existing data, the user passes through all three screen phases. The Update transaction starts when retrieving the existing data, continues to be used as needed to perform field processing, such as lookups, and performs any database updates.

Entering New Data

When a screen is used to enter new data, only the Process and Update phases are involved. The Update transaction is started, if required, during the Process phase (to satisfy lookups during field processing) or to send the new record to the database when the Update phase is initiated.

When updating in QUICK, the records to be updated are locked, re-fetched, and checksummed to ensure that they have not been changed before being updated.

Cursor Retention

By default, PowerHouse retains a read chain beyond a commit for Primary and Detail files, allowing updating along a chain. In some cases, such as ordered retrieval, **ALLBASE/SQL** does not allow cursor retention, and so PowerHouse cannot retain the chain after the transaction has been committed.

The Optimistic Model in QUICK

The Optimistic model is a variation of the Concurrency model that provides higher concurrency for database products that use pessimistic locking on reads within read/write transactions, such as Oracle Rdb. With this model, the trade-off for higher concurrency is reduced ease of development.

This model logically separates screen retrieval and update activities; "read" activities are separated from "write" activities. As a result, reads tend to be non-blocking, but you may not see the most recent data. Consequently, in environments where reference tables are not static, lookups see "stale" data more often. Developing screens which depend on reads being able to see recent changes (including changes made within the same screen) is more difficult using this model.

Note: There is no difference between the default Concurrency and Optimistic models for ALLBASE/SQL, DB2 ODBC, Oracle, or Sybase.

Predefined Transactions

In the Optimistic model, PowerHouse defines two predefined transactions:

- Query transaction
- Update transaction

These are identical to those in the Concurrency model.

A complete chart showing all of the transaction models and associated predefined transactions appears on (p. 49).

The differences between the models lie in the transaction associations. In the Optimistic model:

- all "read" activities are associated with the transaction associated with the Query phase. By default, this is the Query transaction.
- all "write" activities are associated with the transaction associated with the Update phase. By default, this is the Update transaction.

For all database systems other than Oracle Rdb, the Update transaction is used for all activities.

As in the Concurrency model, PowerHouse verifies that data to be updated has not been changed by another user by re-reading and checksumming data prior to updating it.

The Optimistic model is implemented in **Oracle Rdb** using two separate transactions: Query and Update. The Query transaction is a read-only transaction (that is, it will read from the snapshot file if necessary). Snapshotting should be enabled (immediate) in Oracle Rdb; otherwise the Query transaction cannot read from the snapshot file and conflict will result. The Update transaction is a read/write transaction.

By default, QUICK does not reserve any tables for the PowerHouse transactions.

The Consistency Model in QUICK

The Consistency model provides high data consistency at the cost of reduced concurrency.

Predefined Transactions

In this model, there is only one predefined transaction, the Consistency transaction. The Consistency transaction is a read/write transaction with a high isolation level, and is used for all application activities.

A complete chart showing all of the transaction models and associated predefined transactions appears on (p. 49).

Any checking for conflicts among concurrent users' updates is done by the database.

A Consistency transaction has one screen phase associated with it - Consistency.

As a result of the high isolation level used in this model, each user's data is protected from being changed by other users (though not necessarily guaranteeing that a user will be able to update). As a potential side effect of enforcing this level of isolation, database products may protect (lock) more than just the data that has been "touched" by the user; this may diminish other users' ability to access data concurrently.

For more information on isolation levels, see (p. 50).

By default, the transaction is committed at the end of Update processing. As a result, it is generally not possible to browse and update a chain of related data, and commands like "Update Stay" are not supported. The SUBSCREEN statement and the RUN SCREEN verb are not supported if any data on the screen has been committed.

Consistency Model Database Specifics

ALLBASE/SQL

All database activity is associated with one Consistency transaction. By default, PowerHouse uses the ALLBASE/SQL KEEP CURSOR option for Primary and Detail files to allow updating along a chain. This allows PowerHouse to retain a chain beyond a commit. In some cases, such as ordered retrieval, ALLBASE/SQL does not allow its KEEP CURSOR option to be used, and so PowerHouse cannot retain the chain after the transaction has been committed.

DB2

All database activity is associated with one Consistency transaction.

ODBC

All database activity is associated with one Consistency transaction.

Oracle Rdb

All database activity is associated with one Consistency transaction. By default, this transaction is a read/write transaction and reserves no tables.

Oracle

All database activity is associated with one Consistency transaction.

By default, PowerHouse uses the SERIALIZABLE isolation level for transactions defined within the QUICK Consistency model. A Consistency model transaction must lock the rows that it has read so that these rows will not be updateable by another transaction.

For Oracle versions 8i and above, Oracle supports SERIALIZABLE. However, SERIALIZABLE as defined in Oracle does not lock the rows as required by the Consistency model. SERIALIZABLE in Oracle only guarantees that the set of rows read by the transaction will be the same upon reissuing the same SELECT within the same transaction. Without locking applied to the rows read within the transaction, another user could update data read by the consistency transaction.

To implement the locking required for the Consistency model for Oracle, PowerHouse generates the FOR UPDATE clause to SQL SELECT statements generated for Consistency mode screens. Any read locks acquired by the transaction will not be released until the transaction is committed or rolled back.

The following rules govern how the FOR UPDATE clause is added to SQL SELECT statements generated by QDESIGN. These restrictions should be considered by application designers who write their own SQL statements and require a high level of consistency in their applications.

- For SELECT statements that have an associated WHERE clause, the FOR UPDATE is added without any restrictions.
- For SELECT statements that do not have an associated WHERE clause, FOR UPDATE may only be added if all the columns are specified in the SELECT qualifier list.

For more information on issues with the SERIALIZABLE isolation level in Oracle, see (p. 95).

Sybase

All database activity is associated with one Consistency transaction in a dbprocess.

Cursor Retention

By default, in the Consistency Model, the read chain is lost after a commit for all databases except ALLBASE/SQL.

The Dual Model in QUICK

Dual indicates that one transaction model is used for Entry and Find actions, and another is used for Select actions. By default, the Consistency model is used for Entry and Find actions, and the Concurrency model is used for Select actions.

Predefined Transactions

The Dual Model uses three predefined transactions:

- query transaction
- update transaction
- consistency transaction

A complete chart showing all of the transaction models and associated predefined transactions appears on (p. 49).

The default models used in the Dual model can be changed by using the SYSTEM OPTIONS statement in PDL. You can also use the options on the SCREEN statement to override the defaults specified in the dictionary.

If you change the default transaction model in the dictionary to Optimistic, you should also change the default model for Select mode in the Dual model to Optimistic. Otherwise, by default, when the Dual model is used on any QUICK screen, the model used for Select mode will be Concurrency.

Transaction Attributes in QUICK

QUICK supports a variety of transaction attributes on the SCREEN and TRANSACTION statements. Some attributes that QUICK supports are not supported by the underlying database, and vice versa. The behavior of the transaction model depends on the attributes that are supported by QUICK and the target database. This section summarizes and compares some of the transaction attributes supported by QUICK and the supported databases.

For a complete list of the supported transactions attributes, see the TRANSACTION statement, in Chapter 3, "QDESIGN Statements", in the *QDESIGN Reference* book.

Predefined Transactions

QUICK has three predefined transactions:

- Query transaction
- Update transaction
- Consistency transaction

Each transaction model has at least one default predefined transaction associated with it:

Predefined Transactions	Concurrency Model	Optimistic Model	Consistency Model	Dual Model
Query Transaction	÷	÷		÷
Update Transaction	÷	÷		÷
Consistency Transaction			÷	÷

Inherited Transactions

Transactions in QUICK often span screens; that is, a transaction may be started and used for activities on one screen, and continue to be active and used on one or more subscreens. A transaction that has been defined on a higher-level screen is referred to as an inherited transaction, indicating that its attributes have been defined outside the current screen. The predefined PowerHouse transactions, Query, Update, and Consistency, are a special case of inherited transaction, called special inherited, since it is assumed their definitions always exist. As a result, references to the default transactions are always valid.

Commit Options

All models have predefined automatic commit points, points in processing at which PowerHouse will automatically commit or roll back locally active transactions (including inherited transactions). These points are determined by the COMMIT ON option of the SCREEN or TRANSACTION statement in effect for the transaction.

For more information on the COMMIT ON option, see the SCREEN and TRANSACTION statements in Chapter 3, "QDESIGN Statements", in the *QDESIGN Reference* book.

For more information on locally active transactions, see (p. 56).

Transaction Access Types

The transaction access type determines the type of activities that can be performed by a transaction and the type of transaction started in the associated database. By default, the Query transaction is a read-only transaction used for the Query phase, the Update transaction is a read/write transaction used for both the Process and Update phases, and the Consistency transaction is a read/write transaction used for all phases.

Isolation Levels

Isolation Levels are a measure of the degree to which each transaction is isolated from the actions of other transactions. Different database products support different transaction isolation levels - some offer a choice of isolation levels, some provide just one. Low levels of isolation mean that transactions are not well protected from each other; that is, simultaneous transactions may get inconsistent results. Higher levels of isolation generally mean that transactions are better protected from each other. At the highest levels, each transaction may be entirely unaware of changes being made by other transactions.

Lower isolation levels generally allow higher concurrency with a potential loss of consistency, while higher isolation levels provide high consistency but generally result in lower concurrency.

The support available for the various isolation level options offered in QDESIGN depends on the support provided by the underlying database software.

If a database doesn't support a specified isolation level, PowerHouse uses the next available higher isolation level. If a higher level is unavailable, PowerHouse uses the highest available lower level.

When isolation levels are upgraded or downgraded, PowerHouse

- issues a warning message at compile-time for user defined transactions
- does not issue warning messages for the default PowerHouse transactions, or for inherited transactions

The following are some of the terms used to describe isolation levels. The levels are listed from lowest to highest, although the levels are not strictly incremental:

Isolation Level	Description
READ UNCOMMITTED	Allows a transaction to see all changes made by other transactions, whether committed or not. Also known as a "dirty read".
READ COMMITTED	Allows a transaction to read any data that has been committed by any transaction as of the time the read is done.

Isolation Level	Description
STABLE CURSOR	Indicates that while a transaction has addressability to a record (that is, has just fetched it), no other transaction is allowed to change or delete it.
REPEATABLE READ	Allows any data that has been read during a transaction to be re-read at any point within that transaction with identical results.
PHANTOM PROTECTION	Doesn't allow a transaction to see new records, or "phantoms", that did not exist when the transaction started.
SERIALIZABLE	Indicates that the results of the execution of a group of concurrent transactions must be the same as would be achieved if those same transactions were executed serially in some order.

Isolation Levels and Generated SQL Limitation in Oracle

PowerHouse adds a FOR UPDATE clause to all SQL cursor specifications sent to an Oracle database whenever a PowerHouse transaction requests an isolation level of Repeatable Read or higher. By default, transactions used in the Consistency model in QUICK and QTP have this characteristic, as would any designer-defined transactions with an isolation level of Repeatable Read, Phantom Protection, or Serializable.

As a result of Oracle restrictions on the use of the FOR UPDATE clause, the SQL generated by PowerHouse may result in an invalid Oracle specification. For example, the FOR UPDATE clause is not allowed in Oracle if the query includes DISTINCT, GROUP BY, any set operator, or any group function. Using any of these features in a cursor will result in an error message being returned from Oracle, indicating that FOR UPDATE is not allowed in the statement.

To resolve or avoid this situation, ensure that these requests are executed within a transaction with a low isolation level. The default Concurrency or Optimistic models use low isolation levels by default. For further information about setting and overriding transaction isolation levels, see the TRANSACTION statement in the *QDESIGN Reference* book.

Consult your Oracle database documentation for more information about restrictions on the use of the FOR UPDATE clause.

Relational Database Locking

PowerHouse lets the database control locking.

The LOCK verb allows explicit locking of an ALLBASE/SQL or Oracle tables, and is ignored for all other supported databases. The UNLOCK verb is ignored for all databases.

For more information about locking, see the LOCK verb in Chapter 8, "QDESIGN Verbs and Control Structures", in the *QDESIGN Reference* book or consult your database documentation.

Database Specific Transaction Attributes

This table shows the expected behavior when using different database transaction attributes. PowerHouse issues a warning message when you specify options for features that are not supported by the target database.

ODBC: When PowerHouse connects to an ODBC data source, it queries the data source capabilities and tailors its behavior to that data source. One aspect that is determined is the transaction isolation levels that the data source supports.

Transaction Attribute	ALLBASE /SQL	DB2	Microsoft SQL Server	Oracle	Oracle Rdb	Sybase
Isolation levels						

Transaction Attribute	ALLBASE /SQL	DB2	Microsoft SQL Server	Oracle	Oracle Rdb	Sybase
READ UNCOMMITTED	READ UNCOMMITTED	READ UNCOMMITTED (DB2's Uncommitted Read)	READ UNCOMMITTED	READ COMMITTED[2]	READ COMMITTED	Warning[1]
READ COMMITTED	READ COMMITTED	READ COMMITTED (DB2's Cursor Stability)	READ COMMITTED	READ COMMITTED[2]	READ COMMITTED	Warning[1]
STABLE CURSOR	STABLE CURSOR	REPEATABLE READ (DB2's Read Stability)	REPEATABLE READ	READ COMMITTED and locks fetched rows[3]	REPEATABLE READ	Warning[1]
REPEATABLE READ	REPEATABLE READ	REPEATABLE READ (DB2's Read Stability)	REPEATABLE READ	for Read Only transactions - uses Oracle's READ ONLY for Read/Write transaction - uses READ COMMITTED and locks fetched rows	REPEATABLE READ	Warning[1]
PHANTOM PROTECTION	REPEATABLE READ	SERIALIZABLE (DB2's Repeatable Read)	SERIALIZABLE	SERIALIZABLE	SERIALIZABLE	Warning[1]
SERIALIZABLE	REPEATABLE READ	SERIALIZABLE (DB2's Repeatable Read)	SERIALIZABLE	SERIALIZABLE	SERIALIZABLE	Warning[1]
Deferring Constraints	Supported	Warning	Warning	Warning	Warning	Warning
Transaction Priority	Supported	Warning	Warning	Warning	Warning	Warning
Reserving List	Supported[4]	Warning	Warning	Supported	Supported	Warning
Wait for locked database resources	Supported	Warning	[NO]DBWAIT	Supported	Supported	Warning
Read-only Read/write options	Supported	Warning	n/a	Supported	Supported	Supported

Transaction Attribute	ALLBASE /SQL	DB2	Microsoft SQL Server	Oracle	Oracle Rdb	Sybase

Supported: supported both by PowerHouse and the target database.

Warning: a warning message results. The transaction attribute is supported by PowerHouse but not by the target database.

[1] *All locks and lock escalation are managed by Sybase and cannot be overridden.*

[2] *Uses Oracle statement-level read consistency for read/write transactions.*

[3] *Uses Oracle transaction-level read consistency for read-only transactions, and statement- level read consistency for read/write transactions.*

[4] *PowerHouse issues lock table requests for tables in the reserving list.*

Default Transaction Attributes in QUICK

The attributes for the PowerHouse default transactions are:

	Access	Isolation Level	Commit Point	Phase
Query	read-only	READ COMMITTED	MODE [2] NOCOMMIT on subscreens that receive master files	Query
Update (DB2, Sybase)	read/write	REPEATABLE READ	automatic-commit-option[1]	Process, Update, Consistency[3]
Update (ALLBASE/SQL, Microsoft SQL Server, and Oracle)	read/write	READ COMMITTED	automatic-commit-option[1]	Query, Process Update, Consistency
Consistency	read/write	SERIALIZABLE	automatic-commit-option[1]	Query, Process Update, Consistency[4]

[1] *By default, the commit option is taken from the SCREEN statement, which defaults to COMMIT ON UPDATE.*

[2] *For more information on the Query transaction commit timing, see* (p. 56).

[3] *By default, if a table is passed from a screen that is using either the Concurrency or Optimistic model to a screen using the Consistency model, then the Update transaction is used for all operations.*

[4] *By default, if a table is passed from a screen using the Consistency model to a screen that is using either the Concurrency or Optimistic model, then the Consistency transaction is used for all phases.*

The attributes for the PowerHouse default transactions can be changed by using the TRANSACTION statement in PDL or QDESIGN. For more information, see Chapter 2, "PDL Statements", in the *PDL and Utilities Reference* book and Chapter 3, "QDESIGN Statements", in the *QDESIGN Reference* book.

Summary of Relational Models in QUICK

	Concurrency Model	Optimistic Model	Consistency Model
Goal of Model	Provide high multi-user access to data.	Provide higher concurrency for database products that use pessimistic locking on reads within read/write transactions.	Provide high data consistency.
Default Predefined Transactions	Query transaction Update transaction	Query transaction Update transaction	Consistency transaction
Screen phases	Query phase Update phase Process phase	not applicable	Consistency phase
Database access		not applicable	
Finds...	Query phase		Consistency phase
Lookups...	Process phase		Consistency phase
Updates...	Update phase		Consistency phase
Default Transaction Associations	Query phase uses the Query transaction. Process and Update phases use the Update transaction.[1]	Reads use the Query transaction.[2] Writes use the Update transaction.	Consistency phase uses the Consistency transaction.
Checking for Conflicts	Data is not protected during a read; it is only protected when a change is made. When updating, PowerHouse verifies that the data to be updated has not been changed by another transaction, by re-reading and checksumming data prior to updating it.[3]	Data is not protected during a read; it is only protected when a change is made. When updating, PowerHouse verifies that the data to be updated has not been changed by another transaction, by re-reading and checksumming data prior to updating it.[3]	Checking for conflicts among concurrent users' updates is done by the database.
Advantages	Suitable for applications with few "natural" update conflicts between users, or applications that mix data from relational and non-relational sources. Provides for a more current referential integrity check than the Optimistic model against frequently changed tables. Possible to browse a chain of records and commit updates as they are made.	Suitable for use with databases with pessimistic locking on reads. Screen retrieval and update activities are logically separated. Possible to browse a chain of records and commit updates as they are made.	Useful for specialized screens that perform short transactions, or require protection of data prior to update. Each user's data is protected from being changed by other users. One read/write transaction with a high isolation level.

	Concurrency Model	Optimistic Model	Consistency Model
Disadvantages	Complexity	Trade-off for high concurrency is reduced ease of development. No increase in concurrency for most supported databases. It is not uncommon within a single QUICK session to do an update on one screen which provides required information for a lookup on a calling screen. However, because the Query transaction has not been committed since the update was done, the change will not be visible. Changes may be made to tables by other users which, because of the duration of the Query transaction, are not visible when the lookup is done. Consequently, the user may enter invalid information based on an out-of-date lookup.	Concurrency is reduced at the cost of high data consistency; some of the flexibility typical of a PowerHouse screen is lost. Database products may protect (lock) more than just the data that has been "touched" by the user, diminishing other users' ability to access data concurrently. With COMMIT ON UPDATE and ON NEXT PRIMARY options on the SCREEN and TRANSACTION statement, the read chain is lost when you update the primary record.[4] The "Update Stay" command is not valid with COMMIT ON UPDATE.

[1] *For ALLBASE/SQL, the Update transaction is used for all screen phases.*

[2] *For ALLBASE/SQL, the Update transaction is used for all activities.*

[3] *For Sybase, in Browse Mode, PowerHouse first checks if the table accessed contains a timestamp column. If it does, PowerHouse passes all updates to Sybase. Otherwise, PowerHouse re-reads and checksums the row prior to updating it.*

[4] *Not applicable for ALLBASE/SQL, as PowerHouse uses the ALLBASE/SQL KEEP CURSOR option by default to retain the cursor.*

Default Transaction Timing in QUICK

A combination of the automatic activities performed by QUICK and the explicit QDESIGN statements and verbs added by the designer determine when transactions start, commit, or rollback.

- Dictionary SYSTEM options determine the default transaction models.
- At the highest level, the SCREEN statement determines the transaction strategy and automatic commit points used for the screen.
- At the next level, the TRANSACTION, CURSOR, and FILE statements, and the AUTOCOMMIT option on the FIELD statement may be used to customize or override screen defaults for particular transactions, transaction associations, or records.
- At the lowest level, the START, COMMIT, and ROLLBACK verbs provide procedural control over the use of transactions on a screen, either to customize existing behavior or to specify the behavior completely.

- Additionally, the support provided by the underlying database(s) can affect how transactions work.

By default, the Update and Consistency transactions are committed at the end of Update processing. When you enter an update action your changes are sent to the database and made permanent (if a commit is done, a rollback would undo the changes in the database). Updates may be made:

- explicitly with the U, US, UR, or UN action commands
- programmatically, with the AUTOUPDATE option on the SCREEN statement
- with the PUT verb
- with the SQL UPDATE verb

For a description of the syntax used to indicate the commit frequency, see the COMMIT ON options of the SCREEN and TRANSACTION statements in Chapter 3, "QDESIGN Statements", in the *QDESIGN Reference* book.

Locally Active Transactions

A table may be associated with one or more transactions. Only locally active transactions are committed at automatic commit points. A transaction is marked as locally active:

- when it is used to read or write a local table (any table declared on the screen and not passed from a calling screen).
- In return to the calling screen, if the transaction associated with a local table that was passed to a subscreen is still active, and was used to read or write the table on the subscreen.

The use of a transaction name on a COMMIT or ROLLBACK verb is not considered a reference for the purposes of determining whether the transaction is "locally active".

Query Transaction Commit Timing

By default, a Query transaction's commit option is COMMIT ON MODE; the transaction is committed when you change mode, start a new entry or find sequence or leave the screen.

In the Concurrency and Optimistic transaction models, the Query transaction's commit option is NOCOMMIT for subscreens that receive master files. This allows any ongoing read chain to be preserved upon return to the parent screen.

Transaction Timing Example

In Find mode, a record is retrieved on the ORDERS screen:

```
> SCREEN ORDERS TRANSACTION MODEL CONCURRENCY
> FILE ORDERS
> FIELD ORDER_NO OF ORDERS LOOKUP NOTON ORDERS
> SUBSCREEN PARTS PASSING ORDERS
.
.
.
> SCREEN PARTS RECEIVING ORDERS
> FILE ORDERS MASTER
> FILE PARTS PRIMARY
.
.
> FIELD PART_NO OF PARTS LOOKUP NOTON PARTS
```

Query Transaction

In Find mode, the Concurrency model Query phase is active. The Query transaction retrieves the ORDERS information. The Query transaction is locally active since it was used to read a local record. The Query transaction is committed on change of mode on this screen since the ORDERS screen does not receive a master record from any calling screen.

On the PARTS screen, the Query transaction is locally active if the PARTS information is retrieved. However, the Query transaction is not committed on this screen since it receives a master record. This ensures that the retrieval request in the ORDERS parent screen is not terminated due to a commit of the Query transaction on the child PARTS screen.

Update Transaction

On the ORDERS screen, if you change an existing field or use Entry mode to enter a new record which requires a lookup on a relational table (such as the ORDER_NO field), then QUICK uses the Update transaction because you are in the Process phase of the Concurrency model. The Update transaction is locally active because it is used for a lookup on a local record. It is committed when you enter an Update command, change modes, or exit from the screen.

If you enter an order number on the ORDERS screen, and call the PARTS subscreen, the Update transaction is still active and may be used on the PARTS screen. The Update transaction is only considered locally active on the PARTS subscreen if it is used for database activity on the local PARTS record.

Automatic Commit Points

QUICK provides a choice of several automatic commit points. To customize the commit frequency for your application, you specify a commit option on the SCREEN or TRANSACTION statement. For a description of the syntax used to indicate the commit frequency, see the COMMIT ON options of the SCREEN and TRANSACTION statements in Chapter 3, "QDESIGN Statements", in the *QDESIGN Reference* book.

The COMMIT ON options are:

- ON UPDATE
- ON NEXT PRIMARY
- ON MODE
- ON EXIT
- NOCOMMIT

ON UPDATE

This option is the default for the Update and Consistency transactions. Use the ON UPDATE option to ensure that related updates (for example, updates to primary and secondary data) are grouped together, but keep individual transactions relatively short. Committing transactions frequently enables all users to see up-to-date data in their own transactions. Using short transactions reduces the possibility of conflicts with other users, and minimizes the amount of work that a user must repeat in the event of a rollback. The transaction is committed

- when an Update action is completed (before the POSTUPDATE procedure)
- when the screen mode changes (before the PREENTRY and PATH procedures)
- when the user exits the screen (before and after the EXIT procedure)

In the description of when transactions are automatically committed by QUICK, the terms "before" and "after" a procedure can be interpreted as being "at the top of" and "at the bottom of" that procedure, respectively. The intent here is to make it clear that automatic commits are performed by QUICK at various points in the screen processing cycle, whether or not corresponding procedural code exists in the screen definition.

ON NEXT PRIMARY

Use this option if you want to group all detail records (perhaps requiring several entry or display screens) together with primary and secondary records and treat them as a unit to be committed or rolled back.

A screen that includes Order-Header information and an unknown number of Order-Detail records may all be treated as a unit. The transaction is committed when

- the user starts an entry sequence (before the PREENTRY procedure)
- the user retrieves the next set of primary records (before the FIND procedure)
- the user exits the screen (before and after the EXIT procedure)

ON MODE

This option is the default for the Query transaction. Use the ON MODE option to ensure that changes to a series of existing records (for example, all employees in a certain branch or all tasks in a project) are committed or rolled back as a group.

The transaction is committed when

- the screen mode changes (before the PREENTRY and PATH procedures)
- on screen exit (before and after the EXIT procedure)

ON EXIT

Use this option when all activity done on a screen is to be treated as a single transaction. This screen should be one that is exited frequently (such as a single-purpose subscreen called from a main screen), otherwise transactions may extend longer than desirable. Screen exits occur when the user returns to a higher-level screen or leaves QUICK completely. Calling a subscreen or switching threads are not considered screen exits.

An example is a screen in a reservation exchange system, where the screen operator must find the existing reservation, cancel it, and then make a new reservation. These activities might require the operator to find an existing record using the Find action, delete or change it, and finally enter a new record using the Enter action. In this case, the PowerHouse transaction issues several update actions, and changes modes and primary records. By using the COMMIT ON EXIT option, you ensure that all these actions are treated as a unit such that the reservation is canceled and a new one is made, or the original reservation is retained.

The transaction is committed when the screen is exited (after the EXIT procedure).

NOCOMMIT

This option defers the commit to another screen. Use the NOCOMMIT option to ensure that several subscreens (or slave screens) are completed before the PowerHouse transaction is considered complete.

For example, entering information about a new employee might involve filling in numerous pieces of information on several screens. NOCOMMIT could be specified for all but the last of these screens to prevent anybody from committing partial employee information.

The transaction is committed immediately after the EXIT procedure for the top level screen (if no other commit is encountered).

Automatic Commit Points Summary

The following table indicates when a transaction is committed for the various commit points:

Event	ON UPDATE	ON NEXT PRIMARY	ON MODE	ON EXIT
Update complete; before the POSTUPDATE procedure	÷			
New entry sequence before the PREENTRY procedure	÷	÷	÷	
New path determination before the PATH procedure	÷		÷	
Next primary retrieval before the FIND procedure		÷		
Leaving screen before the EXIT procedure	÷	÷	÷	
Return to calling screen after the EXIT procedure	÷	÷	÷	÷

Overriding the Transaction Defaults in QUICK

The following tables summarize the statements, verbs, and options that you can use to override transaction defaults, such as specifying which transaction model to use, when transactions are to be committed, and what activities are grouped in the transaction. As a guideline

1. Assess the application's needs.
2. Choose the most appropriate high-level defaults.
3. Refine using the power of the PowerHouse specification language.
4. Use the procedural level sparingly.

PDL Statement	Purpose/Effect
SYSTEM OPTIONS	Specifies the default transaction model and default definition for the Dual model.
TRANSACTION	Used to override the default transaction characteristics.

QDESIGN Statement	Options	Purpose/Effect
CURSOR, FILE	AUTOCOMMIT	For REFERENCE files, indicates that the transaction used to perform the lookup should be committed as soon as the lookup is completed.
FIELD	LOOKUP, AUTOCOMMIT	Indicates that the transaction used to perform the lookup should be committed as soon as the lookup is completed.
SCREEN	TRANSACTION MODEL	Specifies the transaction model in effect for the screen.
SCREEN, TRANSACTION	COMMIT ON	Determines which automatic commit points are in effect, and provides grouping for activities.
SET	LIST TRANSACTION	Shows the transaction definitions that are in effect for a screen.
TRANSACTION		Defines the attributes of a user-defined transaction, or overrides the attributes of a PowerHouse default transaction.
[SQL] DECLARE CURSOR FOR CALL		Used for calling some types of stored procedures.

Verbs	Options	Purpose/Effect
COMMIT, ROLLBACK, START		Provides explicit transaction control.
[SQL] CALL		Used for calling some types of stored procedures.

Verbs	Options	Purpose/Effect
[SQL] CALL, [SQL] DELETE, [SQL] INSERT, [SQL] UPDATE	TRANSACTION	Indicates which transaction to associate with the statement.

Predefined Conditions	Options	Purpose/Effect
TRANSACTION	IS ACTIVE|INACTIVE IS LOCALLY ACTIVE	May be used to decide whether a transaction should be committed or rolled back.

Program Parameters	Purpose/Effect
commitpoints=obsolete	Overrides the definition of "commit on update" to provide commit timing consistent with earlier versions of PowerHouse.
dbwait|nodbwait	Determines what happens when a requested resource is in use.

QKGO	Purpose/Effect
Rollback Time-out	Controls how long a blocking transaction can stay in the "rollback pending" state.
Rollback Clear	Affects the behavior of screens when an error associated with a database transaction occurs during the Update Phase.

In QDESIGN, the attributes for a transaction are determined as follows:

1. The attributes are set to default values.
2. If the transaction is defined in the dictionary, then the attributes specified in the dictionary are applied, and override any default attributes.
3. If there is a transaction defined for the screen, then the attributes specified on the QDESIGN TRANSACTION statement are applied, and override any attributes defined previously.

Attaches and Transactions in QUICK

PowerHouse manages attaches and transactions to access relational database systems. An attach opens the database and makes the PowerHouse application known to the database. A transaction is used to access the database. All requests to read, insert, update, or delete database information are done by associating the requests with a transaction.

Each relational database system has different capabilities for attaches and transactions.

The following table outlines the different requirements for the supported databases:

Database	Requirement
ALLBASE/SQL	Requires a separate attach for each distinct transaction.
DB2	Does not require a separate attach for each transaction.

Database	Requirement
Microsoft SQL Server	Does not require a separate attach for each transaction.
ODBC	When PowerHouse connects to an ODBC data source it queries the data source capabilities and tailors its behavior to that data source. Each relational database system has different capabilities for attaches and transactions.
Oracle	Requires a separate attach for each distinct transaction.
Oracle Rdb	Does not require a separate attach for each transaction.
Sybase	PowerHouse associates activities with separate Sybase dbprocesses. A single PowerHouse transaction may map to multiple dbprocesses, since a single dbprocess cannot process more than one type of request at a time.

Recycling Attaches

As attaches consume resources, PowerHouse tries to minimize the number of attaches it uses. When a transaction ends either by being committed or rolled back, instead of issuing a detach call, PowerHouse preserves the attach for future use. PowerHouse re-uses an attach to start a new transaction when another attach is needed and the attach is for the right database. A new attach is issued if there are no attaches available or none match.

Consider a QUICK application that uses an Oracle database. When QUICK starts, it needs to retrieve the dictionary information from the database. To do this, QUICK performs the following:

1. Starts the PowerHouse transaction, Dictionary. This requires an attach to the database.
2. Issues an attach since this is the first attach and there are no unused attaches available.
3. Issues the request to retrieve the dictionary information, since a database transaction is associated with the attach.
4. Commits the dictionary transaction since it is no longer required when QUICK has completed the initial request for dictionary information.
5. Commits the underlying database transaction to commit the PowerHouse transaction.

The attach to the database is marked as free for future use.

When you start to access the database, for example, by using Find mode to retrieve data, QUICK performs the following:

1. Starts the PowerHouse Query transaction to retrieve the data. This requires a database transaction which, in turn, requires an attach to the database.
2. Searches the list of attaches for an available attach. In this case, the attach used for dictionary requests is currently available.
3. Retrieves the information since the database transaction is associated with this attach.

If the PowerHouse Update transaction is needed to update a change, and the Query transaction is still active, a second attach is issued for the Update transaction (if required by the underlying database).

Starting Transactions in QUICK

When a PowerHouse transaction is started, QUICK, by default, only starts the database transactions necessary at that time, rather than starting all of the database transactions. This avoids the overhead of attaching to a database that may not be required. Should QUICK need to access other databases associated with that transaction at a later time, the other database transactions may then be started.

The options that you specify when building your QUICK screen impact what database transactions are started when a PowerHouse transaction is started.

For example, the START TRANSACTION verb starts all of the database transactions for the PowerHouse transactions specified with the verb. This may be useful if you want to ensure that an attach occurs for every database that the PowerHouse transaction accesses. It also ensures that database transactions are started at the same time to encompass any data operations on any of these databases.

In addition, if the RESERVING option on the TRANSACTION statement is specified, then all the database transactions necessary to reserve the tables in the reserving list are started when the PowerHouse transaction is started.

Consider the following two screens:

The PARENT Screen

Table T1 and T2 are in Oracle databases, OR1 and OR2 respectively. Table T3 is in a DB2 database, D1.

```
SCREEN PARENT TRANSACTION MODEL CONCURRENCY
FILE T1 IN OR1
FILE T2 IN OR2 REFERENCE
FILE T3 IN D1 REFERENCE
...
FIELD F1 OF T1 LOOKUP ON T2
FIELD F2 OF T2 DISPLAY
FIELD F3 OF T1 LOOKUP ON T3
FIELD F4 OF T3 DISPLAY
SUBSCREEN CHILD
...
```

The CHILD Screen

Table T4 is in an Oracle database, OR3. Table T5 is in a DB2 database, D2.

```
SCREEN CHILD TRANSACTION MODEL CONCURRENCY
FILE T4 IN OR3
FILE T5 IN D2 REFERENCE
...
FIELD F5 OF T4 LOOKUP ON T5
FIELD F6 OF T5 DISPLAY
...
```

The number of attaches and transactions depends on the actions you take.

In Find mode, the logical PowerHouse Update transaction retrieves information on the PARENT screen. Three physical database transactions for the PARENT screen start at the same time:

- an Oracle database transaction attached to OR1 retrieves the fields in table T1.
- an Oracle database transaction attached to OR2 retrieves field F2 in table T2.
- a DB2 database transaction attached to D1 retrieves field F4 in table T3.

If you then go to find information on the CHILD screen, the logical Update transaction is still used but now it's associated with

- a new Oracle database transaction attached to OR3 that retrieves the field F5 in table T4.
- a new DB2 database transaction attached to D2 that displays field F6 in table T5.

The two database transactions started for the CHILD screen are started at the same time, but they were started at a different time than the database transactions for the PARENT screen.

Upon returning to the PARENT screen, if you now correct information that was retrieved, or if you enter new data, the same PowerHouse Update transaction is still associated with the earlier database transactions to do the lookups and updates. For example, QUICK

- uses the two Oracle database transactions attached to OR1 and OR2. The attach to OR2 performs the lookup on T2. The attach to OR1 is done on the assumption that the tables in T1 will eventually need to be updated.
- uses the DB2 database transaction attached to D1 to perform the lookup on field F3 in table T3.

Committing Transactions in QUICK

When a single PowerHouse transaction updates two or more databases, PowerHouse uses a two-phase commit protocol to ensure transaction integrity. Two-phase commit protocol is especially required if one or more of the databases is remote and has a high risk of network or remote system failure. In the majority of cases, there is only one database used in an application and two-phase commit is not as useful.

Two-Phase Commit

The first phase is called the "prepare" phase, the second phase is called the "commit" phase. PowerHouse takes the following steps:

1. It issues database prepare calls for all of the underlying database transactions in the PowerHouse transaction.
2. If any of the prepare calls fail, all the database transactions are rolled back.
3. If all the prepare calls succeed, PowerHouse issues a call to commit the first database transaction.
4. If the first commit call fails, all of the other transactions are rolled back.
5. If the first commit call succeeds, PowerHouse issues commit calls for the rest of the database transactions even if one of the remaining commit calls fail.
6. If any of the prepare or commit calls fail, PowerHouse issues an error message.

Committing Multiple Database Transactions

In the following QUICK screen, table T1 and T2 are in Oracle databases OR1 and OR2 respectively, and table T3 is in aDB2 database D1.

```
SCREEN PARENT TRANSACTION MODEL CONCURRENCY
FILE T1 IN OR1
FILE T2 IN OR2 SECONDARY
FILE T3 IN D1 DETAIL
.
.
.
```

Assume you enter new information or modify existing information and then issue an Update command. QUICK commits the PowerHouse Update transaction if the information is successfully updated. This transaction is made up of three database transactions:

- an Oracle database transaction attached to OR1 for updating table T1
- an Oracle database transaction attached to OR2 for updating table T2
- a DB2 database transaction attached to D1 for updating table T3.

QUICK issues the following calls to prepare and commit these changes:

1. Three calls to prepare the three database transactions. If any call fails, QUICK rolls back all three transactions.
2. If all three prepare calls succeed, QUICK issues an Oracle call to commit the first transaction attached to OR1.
3. If this call succeeds, then QUICK issues an Oracle call to commit the transaction attached to OR2 as well as a DB2 call to commit the transaction attached to D1.
4. If any commit fails, PowerHouse issues an error message.

Tuning Attaches in PowerHouse

You can specify the number of buffers used for attaches to change the size of the cache used to buffer data in memory and provide the ability to tune the performance of your database applications.

Increasing the number of buffers may improve performance (as measured by response time), but will also increase the amount of memory used.

Transaction Error Handling in QUICK

QUICK's transaction rollback processing model for relational databases ensures transaction integrity—that is, a group of related operations is either completely done or completely undone. Where possible, QUICK uses the transaction control options of the database or file system to manage transaction integrity. For file systems that do not have transaction control, QUICK provides its own rollback mechanism.

QUICK's rollback model provides sophisticated control of multiple transactions that may be active on many screens. This section explains how QUICK manages multiple active transactions and how QUICK does rollback of transactions.

Relational Transaction Error Handling Terminology

The following terminology list defines terms that you will encounter in the rest of this chapter.

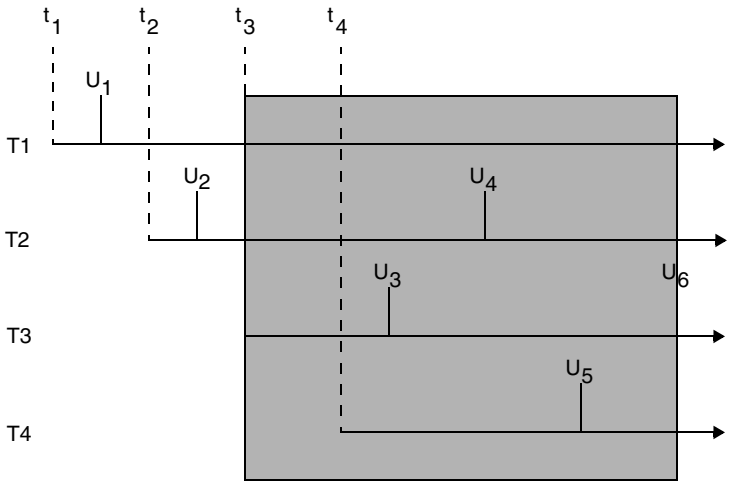
Term	Definition
active transaction	A PowerHouse transaction which has been used to perform a database operation such as data retrieval or update.
ancestor screen	A higher-level screen.
backout	Occurs when data has been changed on a screen and the user changes mode, leaves the screen or has his session timed out before updating.
backout buffer	QUICK backout buffer used to restore Master file buffers on backout.
calling screen	The screen which calls a subscreen.
cascading rollback	The rollback of a conceptual transaction, often requiring that many PowerHouse transactions be rolled back.
conceptual transaction	A conceptual transaction consists of all PowerHouse transactions that have performed any database operations since the time the current active transaction started.
database transaction	A bounded set of database operations that either succeed or fail as a unit.
local record	A table declared on the current screen and not passed from a calling screen.
locally active transaction	An active PowerHouse transaction which has been used to read or write to a local record. An active PowerHouse transaction which has been used to perform any SQL DML command.
PowerHouse transaction	One or more database transactions managed by PowerHouse as a single unit.
rollback buffer	QUICK data buffers used to recover from errors during the Update phase.

Conceptual Transaction

Usually, a single PowerHouse transaction corresponds to a single database transaction. It is possible, though, for a single PowerHouse transaction to consist of several database transactions that might start at different times. Regardless, all the database transactions that make up the PowerHouse transaction will either be committed at the same time or rolled back at the same time.

A conceptual transaction may consist of several individual PowerHouse transactions. The contents of a conceptual transaction are dependent on the screen activity. A conceptual transaction consists of all PowerHouse transactions that have performed any database operations since the time the current active transaction started.

Consider the following example: QUICK is managing four PowerHouse transactions for a particular screen. They are T1 which started at t_1, T2 which started at t_2, T3 which started at t_3 and T4 which started at t_4.



If each U_n represents a database operation, QUICK's view of the conceptual transaction at the time that U_6 in T3 occurs includes transactions T2, T3, and T4. These transactions were all used for database operations after transaction T3 began. T1 is still active at the time of U_6, but there has been no database operation performed in T1 since T3 began. Therefore, T1 is not part of the conceptual transaction.

It is the designer's responsibility to ensure that all PowerHouse transactions comprising a conceptual transaction are treated consistently.

A conceptual transaction may span screen boundaries, such as a PowerHouse transaction that is started or referenced on one screen and used for database operations on a subscreen. When QUICK rolls back, QUICK has to back out of each screen involved in its view of the conceptual transaction.

As the data entered on these screens is not necessarily consistent (because it may be dependent on information retrieved from the database on a transaction that was rolled back), all of the related QUICK buffers may be cleared as well. This occurs for each screen involved in the conceptual transaction. QUICK backs out of each screen until a stable state is reached—that is, the highest level screen where one or more of the rolled back transactions was started or referenced.

This behavior is referred to as "cascading rollback", since the rollback of one PowerHouse transaction may cause other transactions to be rolled back.

Backing Out and Rolling Back

To undo changes to retrieved data or undo entered data on a QUICK screen, you can

- issue a backout command (the default is a caret, ^) from a data field
- use one of the Return commands (Return, Return to Previous Screen, Return to Start, Return to Stop)
- change modes

Each of these actions are ways of backing out.

QUICK issues a warning message if data has been entered or changed but not updated, asking you to repeat the action to confirm. If you repeat the action, QUICK backs out.

Backing out is not necessarily an error condition. If no error has occurred prior to backing out, then QUICK doesn't roll back locally active transactions. In this situation, you should consider the impact on the conceptual transaction, and ensure it is consistent.

Rolling back and backing out are related but different concepts. When QUICK backs out:

- it uses its own backout buffers to undo changes
- it invokes the BACKOUT procedure if there is one
- if QUICK is in a rollback pending state it will attempt to do a rollback. For more information on error handling terminology, see (p. 64).

When Could Rollback Occur?

The purpose of rollback is to undo any changes made by transactions that do not terminate successfully. Rollback may be initiated by the designer using the ROLLBACK verb, or by QUICK in response to an error.

ROLLBACK Verb

Rollback will occur when a ROLLBACK verb is executed.

If the ROLLBACK verb does not specify any transaction name, then all locally active transactions on the screen are rolled back. QUICK performs a cascading rollback if required. If there are no locally active transactions, no rollback is performed.

In the following example, all locally active transactions will be rolled back when the ROLLBACK verb is executed:

```
> PROCEDURE DESIGNER UNDO
> BEGIN
> ...
>          ROLLBACK
> ...
```

If the ROLLBACK verb specifies one or more transaction names, the named transactions are rolled back (even if they are not locally active). QUICK does not perform a cascading rollback in this case; it is up to the screen designer to ensure that any related transactions (that is, other transactions within the conceptual transaction) are rolled back as well.

In the following example, the transaction named MY_UPDATE will be rolled back, regardless of whether it is locally active. No other transactions will be affected:

```
> PROCEDURE DESIGNER UNDO
> BEGIN
>  ...
>     ROLLBACK MY_UPDATE
> ...
```

Errors

Rollback could also occur when an error associated with a transaction is encountered. If a database operation (such as a retrieval, update, or commit) fails, the transaction in which the operation is performed is considered to be in error. Similarly, if an error condition occurs in the application while a transaction is active, that transaction is in error.

Some error conditions within an application are not associated with a transaction and therefore do not affect any transaction. As an example, an error could occur when QUICK is finalizing an item as part of the PUT verb processing. An example of a different situation in which an error would not affect any transaction is described in Case 3 in the Case Studies at the end of this chapter.

QUICK's behavior in the event of an error varies depending on the severity of the error. If the error is at a Severe level, then QUICK will perform a cascading rollback immediately.

If an error is not Severe, then the user can often correct it. In this case, the user would not want his transactions and application work rolled back immediately. QUICK provides two mechanisms designed to allow the user to correct an error, rather than having to re-start the transaction. The mechanism used depends on the value of the QKGO parameter, Rollback Clear. If the parameter is set to Y (the default), then QUICK will keep the transaction in error active and give the user the opportunity to correct the error. This behavior is referred to as Rollback Pending. If the user leaves the screen without correcting the error, or the screen is timed out, then the conceptual transaction will be rolled back using a cascading rollback.

If the Rollback Clear parameter is set to N, then QUICK will attempt to rollback the transaction, without clearing the data on the screen. This behavior is referred to as Rollback Keep Buffers. In some cases, rolling back the transaction would result in the screen's data being inconsistent with the values stored in the database. In these cases, QUICK will not perform Rollback Keep Buffers but instead will revert to Rollback Pending behavior.

Severe Errors

Any of the following error conditions are considered Severe errors:

- QUICK executes a SEVERE verb
- QUICK detects a screen design error (*d* error)
- QUICK detects a design inconsistency (e.g., the file definition in the dictionary does not match the physical file)
- an internal error occurs

Errors

The following error conditions are not considered Severe errors:

- QUICK executes an ERROR verb
- a database error, such as a trigger or constraint violation, occurs

Note: Backing out of a screen is not considered an error condition.

Database Detaches

Rollback can also occur when QUICK attempts to commit active transactions prior to performing a database detach. Transactions that cannot be committed are rolled back.

The flowchart on the next page summarizes QUICK's behavior when an error is encountered or a ROLLBACK verb is executed.

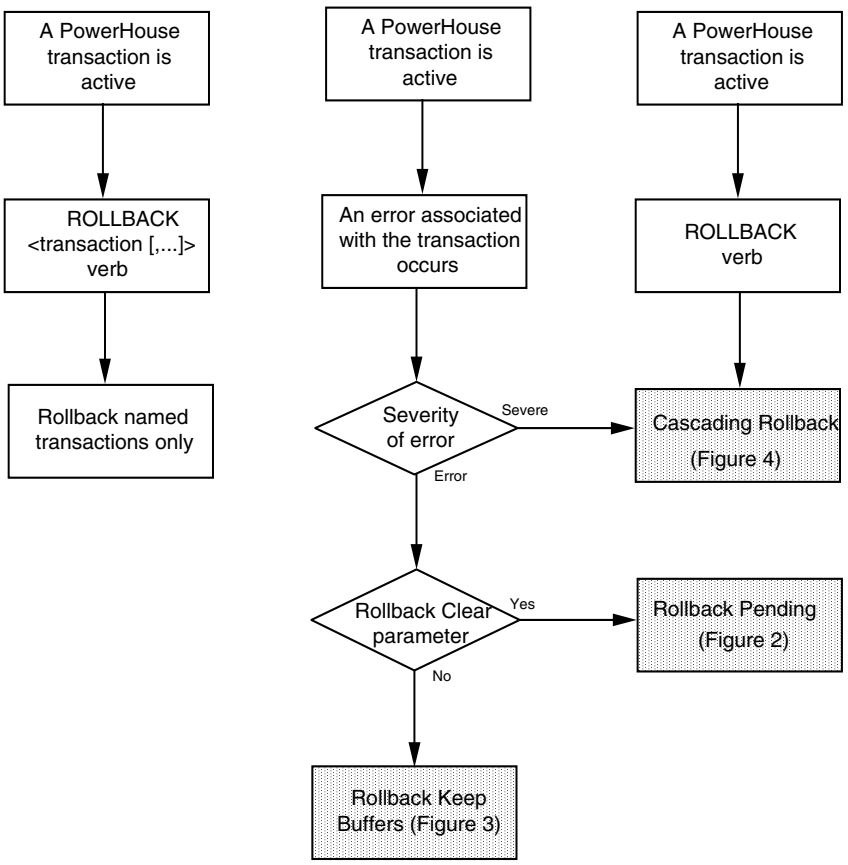


Figure 1: Transaction Rollback Processing

Rollback Pending

When a rollback is pending, the user's ongoing database transactions remain active and all data remains on the screen. The user has an opportunity to correct the error and re-issue the update command. When QUICK is in this state, concurrency may be reduced, as any locks and other resources acquired by the transaction(s) will continue to be held.

The Rollback Time-out QKGO parameter can be used to roll back transactions that could block other transactions from operating. Any read/write transaction is considered a blocking transaction.

When the Rollback Time-out parameter is set to a non-zero value and a rollback is pending, a rollback will occur if there is no terminal activity within the specified number of seconds. A message is issued prior to the timeout to inform the user that the screen will be cleared and a rollback will occur.

If you specify the **clear_rollback_pending** program parameter, the Rollback Pending state is cleared whenever any PowerHouse procedure has completed successfully. Successful procedure completion in any phase (Query, Process, Update, or Consistency) will clear the Rollback Pending state. If you do not use the **clear_rollback_pending** program parameter, successful completion of some types of procedures will not clear the Rollback Pending state.

After the Rollback Pending state is cleared, the Screen is no longer considered to be in error, therefore its transactions can subsequently be committed. Note that the procedure does not actually have to resolve the error condition. The designer must ensure that the condition that caused the error is resolved, otherwise data integrity problems could occur. Once a procedure successfully executes, the screen's transactions can be committed, even in a backout situation. In a Rollback Pending situation, the designer should ensure that the only available actions are to correct the error or backout all changes.

If the user backs out without correcting the error, and there are locally active transactions, then QUICK will roll back the conceptual transaction, using a cascading rollback if necessary.

The following flowchart summarizes the activity during Rollback Pending behavior.

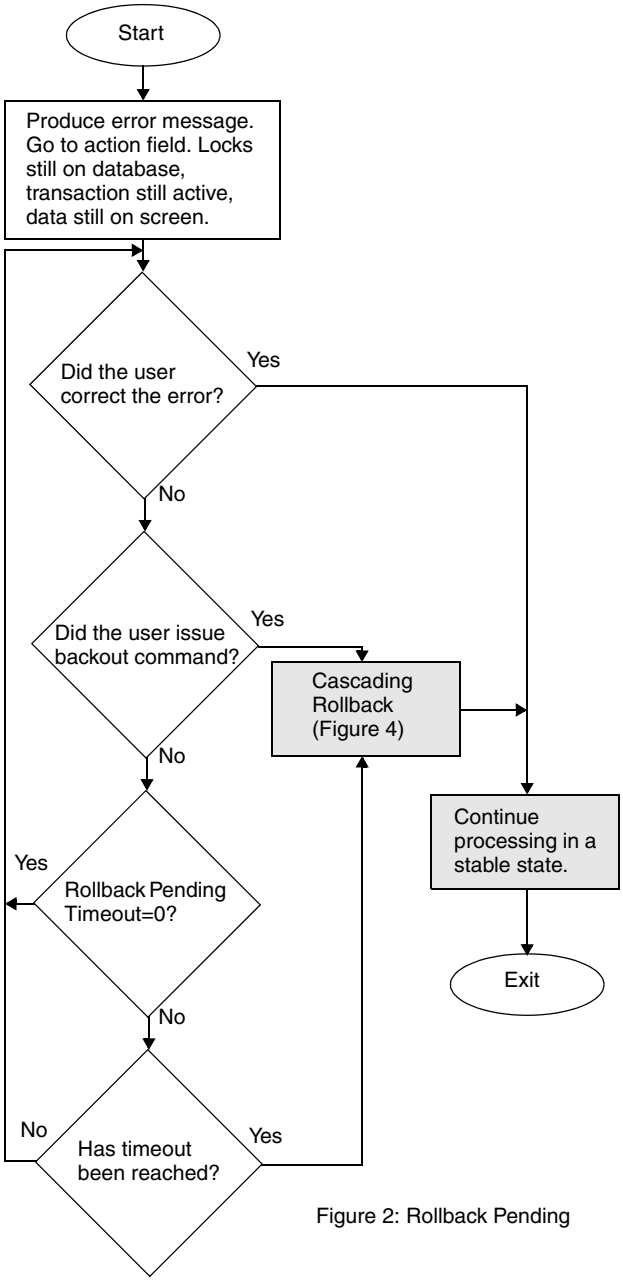


Figure 2: Rollback Pending

In some situations, the user can't correct the problem because the field in error is not on the current screen. In this case, the user may want to provide a DESIGNER procedure that forces an immediate rollback, bypassing the Rollback Pending state. For example:

```
> PROCEDURE DESIGNER UNDO HELP "Undo any data entry"
>   BEGIN
>     ROLLBACK
```

```
>   END
```

The ROLLBACK verb will immediately roll back all locally active transactions.

Rollback Keep Buffers

The Rollback Keep Buffers behavior allows a transaction to be rolled back immediately while the data associated with the transaction remains on the screen. The user can then try to correct the error and re-issue the update. Because the transaction has been rolled back, any database locks acquired by the transaction will have been released. Compare this to Rollback Pending in which the user is given an opportunity to correct the error while the transaction remains active and keeps the resources locked.

As with Rollback Pending, Rollback Keep Buffers behavior only applies when a non-severe error causes the rollback. It is not applicable to Severe level errors or the ROLLBACK verb.

The Rollback Keep Buffers behavior is only possible if the Rollback Clear parameter in QKGO is set to N and certain conditions are met. These conditions are described below and are summarized in Figure 3. For more information on the N action field command, see Chapter 5, "QUICK's Processing Modes", in the *QDESIGN Reference* book.

QUICK can Rollback and Keep Buffers after an error if all of the following conditions are true:

- the QKGO parameter, Rollback Clear has been set to N
- the error occurs during the Update phase of the current Screen
- none of the transactions used during the Update phase were used to write to a table (using a PUT, SQL Delete, SQL Insert, or SQL Update verb) before the Update phase started.
- none of the transactions used during the Update phase have an isolation level higher than read committed
- none of the transactions started before the Update phase have a reserving list specified

In the following example, Rollback Keep Buffers behavior can be used. In this example, two separate transactions are used in the Concurrency model.

```
>   SCREEN RKB TRANSACTION MODEL CONCURRENCY &
>      COMMIT ON UPDATE
>   FILE EMPLOYEES IN Base1 PRIMARY
>   FIELD EMPNO OF EMPLOYEES
>     .
>     .
>     .
> PROCEDURE UPDATE
>     BEGIN
>        PUT EMPLOYEES
>           .
>           .
>           .
>      END
```

If the user issues an Update, the Update transaction is started. If an error is encountered and the Rollback Clear parameter has been set to N, Rollback Keep Buffers behavior can be used. The transaction will be rolled back, and the user's data will continue to be displayed on the screen.

In the following example, however, it may not always be possible to Rollback and Keep Buffers after an error, since the LOOKUP would cause the Update transaction to be active prior to the start of the Update phase:

```
>   SCREEN RKB TRANSACTION MODEL CONCURRENCY &
>      COMMIT ON UPDATE
>   FILE EMPLOYEES IN Base1 PRIMARY
>   FIELD EMPNO OF EMPLOYEES LOOKUP NOTON EMPLOYEES
>      .
>      .
>      .
>   PROCEDURE UPDATE
>      BEGIN
>         PUT EMPLOYEES
>            .
```

```
>            .
>            .
>      END
```

During data entry, or in Change mode after a record has been found, the screen is in the Process phase. When the LOOKUP NOTON for EMPLOYEES is performed, the Update transaction starts and remains active. If an error is encountered on updating, Rollback Pending behavior will be used, regardless of whether the Rollback Clear parameter has been set. In this case, Rollback Keep Buffers behavior cannot be used, since the Update transaction was active before the Update phase began.

For Rollback Keep Buffers behavior to be possible, the transaction(s) used in the Update phase cannot have started prior to the Update phase. If you want Rollback Keep Buffers behavior to be possible in situations such as that shown in the previous example, you have a number of options:

1. Use the AUTOCOMMIT option with the LOOKUP option on the FIELD statement.
2. For databases where two transactions are used in the default Concurrency and Optimistic models, use the Optimistic model.
3. Include Process phase activities in a separate transaction. For example:

```
>   TRANSACTION NEW_QUERY READ ONLY
>   FILE EMPLOYEES IN Base1 PRIMARY &
>      TRANSACTION NEW_QUERY FOR QUERY, PROCESS
```

Any of these options can be used to ensure that the Update transaction is not started prior to the Update phase.

Calling Subscreens in the Update Phase

Certain considerations apply to subscreens called during the Update phase when Rollback Keep Buffers behavior is possible. The subscreens affected are those where processing is logically an extension of the Update phase of the calling screen, that is, subscreens where the following conditions are all true:

- the NOCOMMIT option is in effect
- there is no user interaction or screen display
- control is immediately returned to the calling screen (e.g., via a RETURN verb or the AUTORETURN screen option)

If an error associated with a transaction occurs in the INITIALIZE procedure on this type of subscreen, then Rollback Keep Buffers behavior is possible. Control will return to the calling screen, and the user can re-try the update without having to re-retrieve or re-enter data.

If an error occurs during any other procedure on this type of subscreen, the user will be prompted on the subscreen. This prompting is considered to be user interaction, and in these instances, Rollback Keep Buffers behavior on the calling screen will not be possible.

If an error occurs after the subscreen has completed successfully and control has returned to the calling screen, Rollback Keep Buffers behavior will be possible on the calling screen.

For additional information about error handling and rollback on subscreens, see (p. 75).

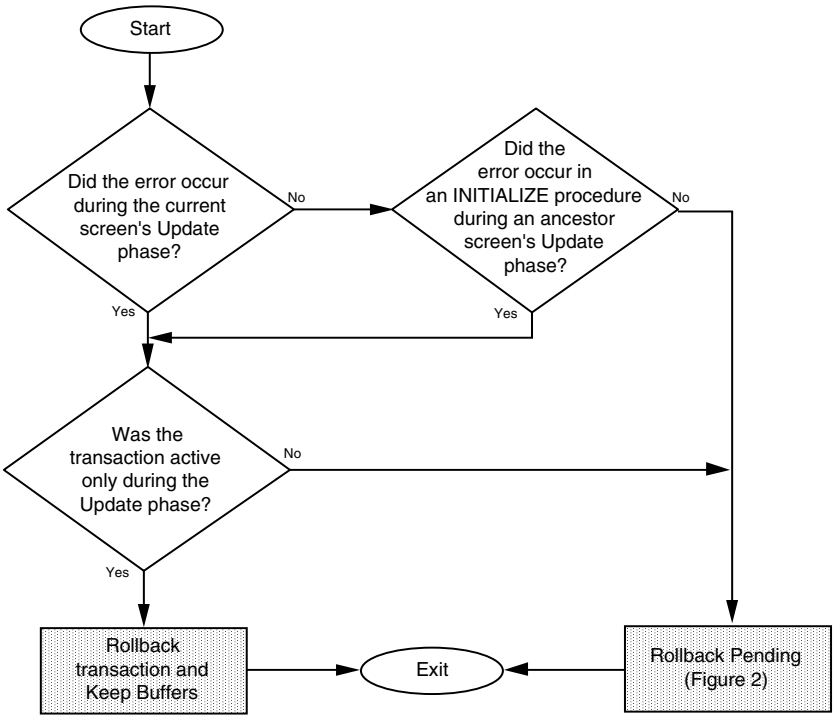


Figure 3: Rollback Keep Buffers

Cascading Rollback

Whenever a rollback is required, QUICK's goal is to return the application to a stable and consistent state. QUICK achieves this by rolling back the conceptual transaction. The term "cascading rollback" is used to describe QUICK's general rollback behavior, since the rollback of a conceptual transaction may require rolling back multiple interdependent PowerHouse transactions, and returning the application to a higher screen level.

The steps involved in performing a cascading rollback are described in the following pages.

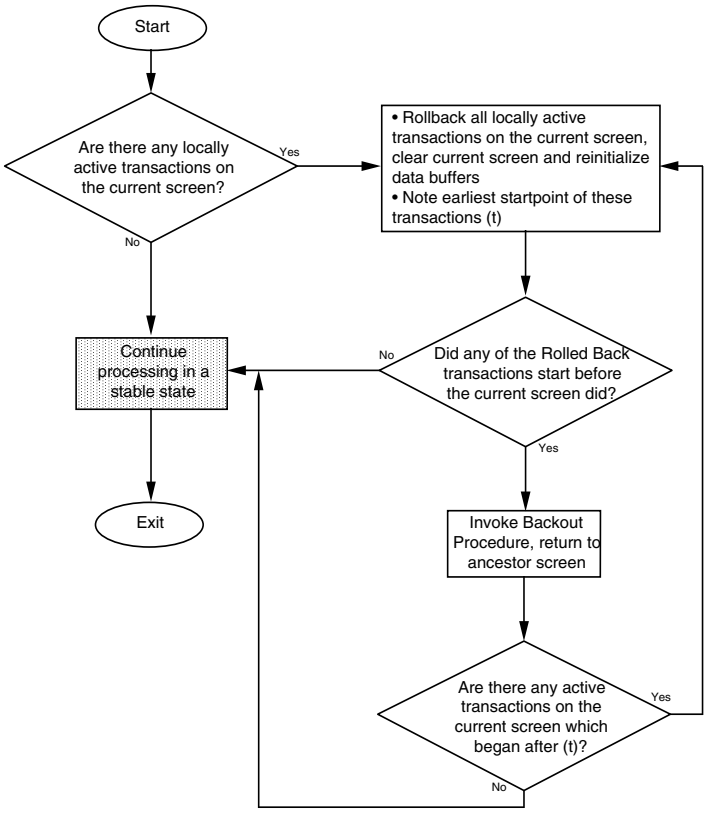


Figure 4: Cascading Rollback

To perform a cascading rollback

1. QUICK determines if there are any locally active read/write transactions on the current screen. If there are none, no rollback occurs and QUICK skips to step 9.
2. If there are locally active read/write transactions, QUICK rolls back these transactions and keeps track of the earliest start point of the rolled back transactions. It determines whether the rolled back transactions were used for retrievals or updates on any ancestor screens.
3. It examines all active read/write transactions started on ancestor screens to determine if any retrievals or updates have been done since the earliest start point determined in step 2.
4. If no retrievals or updates have occurred since this point and all of the rolled back transactions were only used on the current screen or descendant screens, then the rollback is local to the current screen. QUICK clears the data buffers, issues a rollback complete message and skips to step 9.
5. If the rollback is not local to the current screen, then QUICK has to step back through the ancestor screens until a stable state is reached, that is, all rolled back transactions were started on that screen or on one of its subscreens.

   QUICK tries to avoid rolling back the read-only transactions since they are primarily used for the read chain. If the rollback is local to the current screen, read-only transactions are not rolled back, thus ensuring the read-chain is preserved. However, if the rollback is not local, then QUICK assumes that the read-chain is no longer valid and rolls back the locally active read-only transactions.
6. QUICK invokes the BACKOUT procedure for the current screen and clears any of the data buffers for the current screen. QUICK then returns to the calling screen.
7. QUICK marks as locally active any of the active transactions associated with local records on the calling screen that were used on the subscreen for retrievals, updates, and any transactions associated with any SQL DML verbs.
8. QUICK determines if the rollback is local to this screen or requires returning to an ancestor screen. To do this, it follows steps 1 through 7 until it finds a screen where the rollback is local to that screen.

9. At this point, QUICK has reached a stable state. The highest-level screen involved in the rollback has been reached. All transactions that were active in any subscreen are now rolled back.

   All of the read/write transactions that have done retrievals or updates after the earliest start point have been rolled back; this is true whether they were used on the current screen or a descendant screen. There may still be active read/write transactions, but any retrievals or updates done on these transactions were done before the earliest start point.

Rolling back and backing out are related but different concepts. Backing out is used to undo changes when a user decides not to perform an Update. Backing out is not an error condition.

Backout

Prior to update, the user can undo changes to data on a QUICK screen by

- issuing a backout command (the default is a caret, ^) from a data field
- using one of the Return commands (Return, Return to Previous Screen, Return to Start, Return to Stop)
- changing modes

When QUICK backs out:

- it uses its backout buffers to undo changes to master files
- it invokes the BACKOUT procedure if there is one
- if a rollback is pending, QUICK will attempt to do a rollback

When the user backs out from a screen on which changes have been made but not yet updated, QUICK issues the following message:

```
Data has been changed but not updated. Repeat the action if this is OK.
```

If the action is repeated, QUICK clears the data buffers and executes the BACKOUT procedure if there is one.

If no error has occurred prior to backing out, then QUICK doesn't rollback locally active transactions. If the conceptual transaction requires that a rollback occur when a user backs out of a screen, either add a BACKOUT procedure that does a rollback or add a ROLLBACK verb to the existing BACKOUT procedure.

```
>   PROCEDURE BACKOUT
>      BEGIN
>         ROLLBACK
>      END
```

The ROLLBACK verb in this example only performs a rollback of locally active transactions.

In the following EMPLOYEE_DETAIL screen, the EMPLOYEES file, which is passed to the subscreen, is the only file on this screen. By default, the Update transaction is associated with the master file and consequently will not be locally active on this screen. The BACKOUT procedure ensures that any updates done on the Update transaction are undone when you back out.

```
>   SCREEN EMPLOYEE_DETAIL RECEIVING EMPLOYEES &
>      TRANSACTION MODEL CONCURRENCY
>   .
>   .
>   .
>   FILE EMPLOYEES IN PERSONNEL MASTER
>   .
>   .
>   .
>   PROCEDURE BACKOUT
>      BEGIN
>         IF TRANSACTION UPDATE IS ACTIVE
>            THEN ROLLBACK TRANSACTION UPDATE
>      END
```

Backing Out of a Subscreen

When backing out of a subscreen, backout buffers are used to restore master files when there have been changes made to non-master files on the subscreen. Each master file record buffer will be restored to the most recent of state of the master file

- at the start of the subscreen.
- after the last successful PUT that has not been rolled back.

Subscreens and Rollback

Non-Locally Active Transactions in Error

By default, only locally active transactions are ever committed or rolled back on any screen.

If an error associated with a non-locally active transaction occurs, QUICK will stop processing and give the user an opportunity to correct the error. Unless there is some locally active transaction, no commit or rollback will occur by default on the current screen. If there is no locally active transaction, regardless of whether the user corrects the error, non-locally active transactions will not be rolled back on the current screen.

Since QUICK considers that all error conditions are associated with the current screen (the screen on which the error occurred), it will not propagate the error condition to any higher-level screen. When the user returns to the calling screen, the RUN SCREEN verb or SUBSCREEN statement will be treated as if it succeeded, and processing will continue on the calling screen.

UNIX: This is a change in behavior from versions of PowerHouse prior to 7.33.C. Prior to PowerHouse 7.33.C, an error on a PUT to a non-local record was considered an error condition on the ancestor screen where the record was a local record, rather than as an error on the current screen. As a result, locally active transactions could be committed on the current screen, even though an error had occurred. The current behavior resolves this problem, and should reduce the number of situations in which a transaction is committed with an uncorrected error.

Designers must still consider the situation in which an error occurs on a non-locally active transaction on a subscreen. For example,

```
>   SCREEN MAIN
>   FILE RECORD_A IN DATABASE PRIMARY
>   .
>   .
>   .
>   SUBSCREEN SUB1 PASSING RECORD_A
>   BUILD
>   SCREEN SUB1 RECEIVING RECORD_A SLAVE
>   FILE RECORD_A IN DATABASE MASTER
>   .
>   .
>   .
>          PUT RECORD_A
```

As described above, if there is an error on the PUT of RECORD_A on screen SUB1, screen SUB1 will be considered to be in error. However, since there is no locally active transaction on this screen, no rollback will be done. The error will not be propagated to the calling screen, and the transaction could be committed on screen, MAIN. If this is undesirable, you may want to use a flag to check whether an error occurred on the lower screen in order to react appropriately on the calling screen.

Note: A SLAVE screen was used in the example since it is a simple example of a screen in which there would be no locally active transactions. This is not the only type of screen in which this situation could exist.

Here is an example to illustrate the effect of the change in behavior:

```
> SCREEN MAIN
> FILE RECORD_A IN DATABASE PRIMARY
> .
> .
> .
```

```
>  SUBSCREEN SUB1 PASSING RECORD_A
>  BUILD
> SCREEN SUB1 RECEIVING RECORD_A
> FILE RECORD_A IN DATABASE MASTER
> FILE RECORD_B IN DATABASE PRIMARY
> .
> .
> .
>      PUT RECORD_B
>      PUT RECORD_A
> .
> .
> .
```

UNIX: In versions prior to PowerHouse 7.33.C, if an error occurred on the PUT of the master file RECORD_A, on screen SUB1, the transaction could still be committed because the error condition would be associated with screen MAIN. As a result, RECORD_B would be changed but RECORD_A would not be changed, possibly causing a data-integrity problem.

Currently, if an error occurs on the PUT of RECORD_A then screen SUB1 is considered to be in error, and the locally active transaction cannot be committed until the error is corrected.

Locally Active Transactions in Error

If a locally active transaction is in an error state on a subscreen, that transaction can be rolled back on the subscreen. The error state will not be passed back to the calling screen, thus allowing the calling procedure to continue regardless of the error on the lower screen.

If the error message is displayed while control is on the subscreen, the user will have an opportunity to correct the error. If the error is not corrected, the transaction will be rolled back and, on return to the calling screen, the RUN SCREEN verb or SUBSCREEN statement will be treated as if it succeeded, since the user has seen the result of its execution.

UNIX, Windows: If the error is not displayed to the user while control is on the subscreen, the RUN SCREEN verb or SUBSCREEN statement will be treated as if it failed. By default the processing in the calling screen will cease. Prior to PowerHouse 7.33.C, the default behavior was to continue processing regardless of the error on the subscreen. To override the default behavior, or ensure compatibility with previous versions, the ON ERROR option of the RUN SCREEN verb and SUBSCREEN statement can be used.

While the use of ON ERROR CONTINUE will provide compatibility with previous versions, this can result in data integrity problems, as the following example illustrates:

```
>  SCREEN ORDER_CAPTURE
>  FILE ORDERS IN PERSONNEL PRIMARY
>  FILE STOCK_ON_HAND IN PERSONNEL SECONDARY
>  FIELD ORDER_NO OF ORDERS LOOKUP NOTON ORDERS
>  .
>  .
>  .
>  PROCEDURE UPDATE
>     BEGIN
>        PUT ORDERS
>        RUN SCREEN STOCK_CALC ON ERROR CONTINUE &
>           PASSING ORDERS, STOCK_ON_HAND
>        PUT STOCK_ON_HAND
>     END
>  BUILD
>  SCREEN STOCK_CALC &
>     RECEIVING ORDERS, STOCK_ON_HAND
>  ; This screen has no fields or titles and therefore
>  ; is not seen by the user
>  FILE ORDERS IN PERSONNEL MASTER
>  FILE STOCK_ON_HAND IN PERSONNEL MASTER
>  FILE STOCK_ON_ORDER IN PERSONNEL PRIMARY
>  .
>  .
>  .
>  PROCEDURE INITIALIZE
```

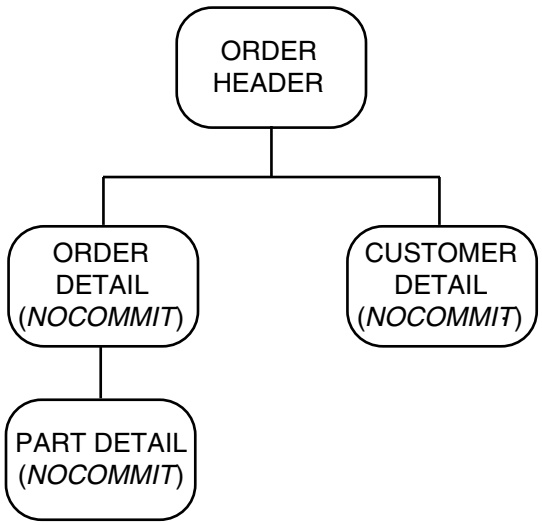
```
>      BEGIN
>          .....
>          PUT STOCK_ON_ORDER
>          IF ......
>              THEN ERROR "The stock figures do not balance"
>          RETURN
>      END
```

Suppose the user enters a new ORDERS record and then issues an Update on the ORDER_CAPTURE screen. The Update transaction is started when the LOOKUP NOTON for the ORDER_NO field is executed. The PUT to ORDERS in the UPDATE procedure is included in this Update transaction. On the subscreen, STOCK_CALC, the PUT to STOCK_ON_ORDER uses the same Update transaction. Since STOCK_ON_ORDER is a local record, the Update transaction is locally active on the subscreen.

Suppose the condition in the INITIALIZE procedure on the subscreen is true and the error occurs. The user will not see the error message while control is on the subscreen because all processing is done in an INITIALIZE procedure and there are no fields or titles on this screen. The user does not have an opportunity to correct the error on the subscreen, and the Update transaction will be rolled back. Processing will continue on the calling screen because of the ON ERROR CONTINUE option on the RUN SCREEN verb. The PUT to STOCK_ON_HAND will be performed, resulting in data integrity problems since the PUTs to ORDERS and STOCK_ON_ORDER have been rolled back.

Rolling Back Through a Screen Hierarchy

When QUICK backs out of a screen hierarchy, it moves straight up the hierarchy, and not through the sibling screens that may have been traversed. In the following Order Entry screen system, consider what happens when the read/write transaction, Update, is started on the main screen, ORDER HEADER, and is used on all the subscreens.



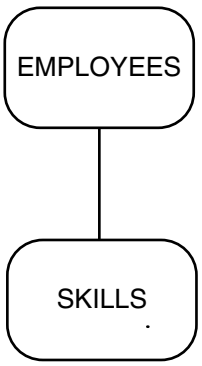
The user first goes to the CUSTOMER DETAIL screen, and then to the ORDER DETAIL screen followed by the PART DETAIL screen. Assume this Update transaction needs to be rolled back because of an error that occurred on the PART DETAIL screen. QUICK backs out of the PART DETAIL screen and the ORDER DETAIL screen and returns to the ORDER HEADER screen. The rollback for the Update transaction is completed at that point and the data buffers for the ORDER HEADER screen are cleared.

Although the CUSTOMER DETAIL screen was not backed out, any updates on the Update transaction are undone as part of the rollback. Any BACKOUT procedure on the CUSTOMER DETAIL screen is not processed.

If there are additional transactions used on the CUSTOMER DETAIL screen that are still active when QUICK returns to the ORDER HEADER screen, they are rolled back as part of the cascading rollback of this screen.

Rollback Case Studies

Consider the following two screens:




```
> SCREEN EMPLOYEES TRANSACTION MODEL CONCURRENCY
> FILE EMPLOYEES IN PERSONNEL PRIMARY
> FIELD EMPLOYEE_NO LOOKUP NOTON EMPLOYEES
> .
> .
> .
> SUBSCREEN SKILLS PASSING EMPLOYEES
```

```
> SCREEN SKILLS TRANSACTION MODEL CONCURRENCY &
>               RECEIVING EMPLOYEES
> FILE EMPLOYEES IN PERSONNEL MASTER
> FILE SKILLS IN PERSONNEL PRIMARY
```

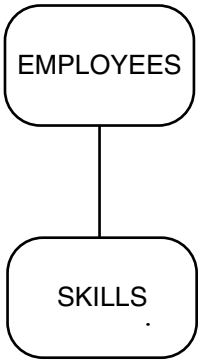
The user enters employee information on the EMPLOYEES screen and then calls the SKILLS screen to enter the skills information for that employee. When the user updates the lower level screen, the Update transaction is used to update both the EMPLOYEES record and the SKILLS record.

If, at the end of the UPDATE procedure, a rollback occurs, then the Update transaction is rolled back, backing out the data entered for the EMPLOYEES record and SKILLS record. Because the Update transaction started on the parent screen (when the lookup was done), QUICK backs out of the SKILLS screen and returns to the EMPLOYEES screen. The screen is then cleared of all data and a rollback complete message is issued. The value of the Rollback Clear parameter in QKGO has no effect in this case since the Update transaction began before the Update phase of the parent screen.

If the Update transaction had started on the SKILLS screen instead, QUICK would remain on the SKILLS screen after rolling back the Update transaction. If the Update transaction began before the Update phase of the SKILLS screen, the data on the screen will be cleared regardless of the setting of the Rollback Clear parameter in QKGO. If, however, the Update transaction began in the Update phase, Rollback Keep Buffers behavior will be possible if the Rollback Clear parameter in QKGO is set to N.

In the following case studies, the SKILLS screen is called from the EMPLOYEES screen to update skills information for an employee. By default, the UPDATE procedure includes PUTs to both the SKILLS record and the EMPLOYEES record.

This example uses the Optimistic model. In this example, assume that the database requires two transactions for the Optimistic model. Lookups are done on the Query transaction, not the Update transaction.




```
>   SCREEN EMPLOYEES TRANSACTION MODEL OPTIMISTIC
>   FILE EMPLOYEES IN PERSONNEL PRIMARY
>   FIELD EMPLOYEE_NO OF EMPLOYEE &
>      LOOKUP NOTON EMPLOYEES
>   .
>   .
>   .
>   SUBSCREEN SKILLS PASSING EMPLOYEES
>   SCREEN SKILLS RECEIVING EMPLOYEES &
>      TRANSACTION MODEL OPTIMISTIC &
>      NOCOMMIT
>
>   FILE EMPLOYEES IN PERSONNEL MASTER
>   FILE SKILLS IN PERSONNEL PRIMARY OCCURS 10
>   .
>   .
>   .
>   PROCEDURE UPDATE
>      BEGIN
>        PUT EMPLOYEES
>        FOR SKILLS
>          BEGIN
>            PUT SKILLS
>          END
>      END
>   .
>   .
>   .
```

The user enters employee information, calls the SKILLS subscreen to enter the skills information for an employee, and issues an Update command. In the UPDATE procedure, QUICK uses the Update transaction to update the EMPLOYEES and SKILLS information.

Case 1: Failure on PUT to Local Record

In this case study, it is assumed that the Rollback Clear parameter in QKGO is set to the default, Y.

If the PUT for the EMPLOYEES record succeeds, but a database error occurs when trying to update the SKILLS record, QUICK will be in a Rollback Pending state. The user can attempt to correct the problem (for example, remove a duplicate skill for that employee), or back out of the screen.

If the user corrects the problem, then the Rollback Pending state is cleared. However, if another error occurs, QUICK returns to the Rollback Pending state.

If the user backs out of the screen while in Rollback Pending state, then QUICK attempts to do a rollback. Since there is a locally active transaction in an error state (the Update transaction was used to unsuccessfully update the SKILLS record), QUICK rolls back the Update transaction and clears the data buffers for the SKILLS record. Since the Update transaction did not start on the parent screen, the rollback will be local to the SKILLS screen. As a result, the Query transaction remains active and the record buffers for the EMPLOYEES record remain intact.

Case 2: Failure on PUT to Received Record

In this case study, it is assumed that the Rollback Clear parameter in QKGO is set to the default, Y.

If a database error occurs when QUICK is executing the PUT verb for the EMPLOYEES record, then QUICK will be in a Rollback Pending state. The transaction is in an error state, but is not locally active.

If the user corrects the problem, then the Rollback Pending state is cleared. The user can re-issue the UPDATE command. However, if another database error occurs on the PUT of EMPLOYEES, QUICK returns to a Rollback Pending state. If the user issues a return command to leave the screen, no rollback occurs since there is no locally active transaction. The transaction error state is not returned to the EMPLOYEES screen. On the EMPLOYEES screen, the procedure that was used to run the screen continues processing.

Case 3: UPDATE Procedure Fails When Database Operation Succeeds

In this case study, it is assumed that the Rollback Clear parameter in QKGO is set to the default, Y.

The EMPLOYEES screen is the same as the original screen, but the SKILLS subscreen is slightly different:

```
> SCREEN SKILLS RECEIVING EMPLOYEES &
>               TRANSACTION MODEL OPTIMISTIC &
>               NOCOMMIT
> FILE EMPLOYEES IN PERSONNEL MASTER
> FILE SKILLS IN PERSONNEL PRIMARY OCCURS 10
.
.
.
> PROCEDURE UPDATE
> BEGIN
>      PUT EMPLOYEES
>      FOR SKILLS
>      BEGIN
>          IF SKILL OF SKILLS NE "POWERHOUSE" &
>             AND SKILL OF SKILLS NE "COBOL" &
>             AND SKILL OF SKILLS NE "FORTRAN"&
>             AND SKILL OF SKILLS NE "PASCAL" &
>             AND SKILL OF SKILLS NE "C"
>          THEN
>             ERROR "Unrecognized programming language"
>          PUT SKILLS
>      END
> END
```

If the PUT to the EMPLOYEES record is completed successfully, and the error condition occurs, QUICK issues the error message, skips the rest of the UPDATE procedure and sets the Rollback Pending state. However, if the error occurs before the first SKILLS record is PUT, then the error cannot be associated with a transaction and, therefore, no transaction will be in an error state.

If the user backs out of the screen by issuing a return command, then QUICK returns to the EMPLOYEES screen, and the Update transaction is not in an error state. The PUT done on the subscreen is committed even if the user backs out of the EMPLOYEES screen.

If an ERROR verb is issued after the PUT to SKILLS, then QUICK is in a Rollback Pending state. If the user backs out, then the Update transaction is rolled back since it is locally active.

Case 4: Interrelated Transactions

In this case study, it is assumed that the Rollback Clear parameter in QKGO is set to the default, Y.

This case illustrates the effect of a cascading rollback on related transactions. (It is not, however, an example of good design.) The screen hierarchy is the same as the Order-Entry System earlier in this chapter.

The ORDER_HEADER screen is the top-level screen for entering order information. It has a subscreen for updating the customer information and another for entering order detail information. In this example,

- TRANS_ORDER_CONTROL is used to update the ORDER_NUMBER control file.
- Part information updates are done on TRANS_PART_UPDATE.
- This procedure creates unique order numbers and part numbers.

```
> SCREEN ORDER_HEADER TRANSACTION MODEL CONCURRENCY
>
>
> TRANSACTION TRANS_ORDER_CONTROL READ WRITE
> TRANSACTION TRANS_PART_UPDATE READ WRITE
>
> FILE ORDER_HEADER IN PERSONNEL PRIMARY
> FILE CUSTOMER_DETAIL IN PERSONNEL REFERENCE
> FILE ORDER_CONTROL IN PERSONNEL DESIGNER &
>                    TRANSACTION TRANS_ORDER_CONTROL
> FILE PART_CONTROL  IN PERSONNEL DESIGNER &
>                    TRANSACTION TRANS_PART_UPDATE
>
> TEMPORARY T_PART_NUMBER NUMERIC
>
> FIELD ORDER_NUMBER OF ORDER_HEADER DISPLAY
> FIELD CUSTOMER_NUMBER OF ORDER_HEADER &
>                    LOOKUP ON CUSTOMER_DETAIL
.
.
.
> SUBSCREEN ORDER_DETAIL PASSING ORDER_HEADER, &
>                                T_PART_NUMBER
> SUBSCREEN CUSTOMER_DETAIL
.
.
.> PROCEDURE PREENTRY
>   BEGIN
>     GET ORDER_CONTROL SEQUENTIAL
>     LET ORDER_NUMBER OF ORDER_CONTROL = &
>         ORDER_NUMBER OF ORDER_CONTROL + 1
>     LET ORDER_NUMBER OF ORDER_HEADER = &
>         ORDER_NUMBER OF ORDER_CONTROL
>     DISPLAY ORDER_NUMBER
>     PUT ORDER_CONTROL
>     GET PART_CONTROL SEQUENTIAL
>     LET PART_NUMBER OF PART_CONTROL = &
>         PART_NUMBER OF PART_CONTROL + 1
>     LET T_PART_NUMBER = PART_NUMBER OF PART_CONTROL
>     PUT PART_CONTROL
>   END
```

The CUSTOMER_DETAIL screen is used for adding or updating customer information. No commits are done on this screen. It uses the TRANS_CUST_UPDATE transaction so that updates to customer information are done in isolation from updates to the order information. In the following example,

- All updates to customer information are done using the TRANS_CUST_UPDATE transaction.
- Since no phases are specified, this transaction will be used for all phases.

```
> SCREEN CUSTOMER_DETAIL NOCOMMIT &
>        TRANSACTION MODEL CONCURRENCY
>
```

```
> TRANSACTION TRANS_CUST_UPDATE READ WRITE
> FILE CUSTOMER_DETAIL IN PERSONNEL PRIMARY &
>                     TRANSACTION TRANS_CUST_UPDATE &
>                     FOR PROCESS,UPDATE
> FILE CUSTOMER_CONTROL IN PERSONNEL DESIGNER &
>                    TRANSACTION TRANS_CUST_UPDATE
>
> FIELD CUSTOMER_NUMBER
.
.
.
> PROCEDURE PREENTRY
>   BEGIN
>     GET CUSTOMER_CONTROL SEQUENTIAL
>     LET CUSTOMER_NUMBER OF CUSTOMER_CONTROL = &
>         CUSTOMER_NUMBER OF CUSTOMER_CONTROL + 1
>     LET CUSTOMER_NUMBER OF CUSTOMER_DETAIL = &
>         CUSTOMER_NUMBER OF CUSTOMER_CONTROL
>     PUT CUSTOMER_CONTROL
>   END
```

The ORDER_DETAIL screen is used for entering order detail information. No commits are done on this screen. It is important to ensure that all ORDER_DETAIL records have a corresponding ORDER_HEADER record. Consequently, the ORDER_DETAIL screen shares the Update transaction with the ORDER_HEADER screen and uses the NOCOMMIT option on the SCREEN statement to ensure that the header and detail records are committed together.

```
> SCREEN ORDER_DETAIL RECEIVING ORDER_HEADER, &
>                     T_PART_NUMBER &
>                     NOCOMMIT &
>                     TRANSACTION MODEL CONCURRENCY
>
> TEMPORARY T_PART_NUMBER NUMERIC
> FILE ORDER_HEADER IN PERSONNEL MASTER
> FILE ORDER_DETAIL IN PERSONNEL PRIMARY OCCURS 10
> FILE PART_DETAIL  IN PERSONNEL REFERENCE
>
.
.
.
> FIELD PART_NUMBER OF ORDER_DETAIL &
>                    LOOKUP ON PART_DETAIL
>
> SUBSCREEN PART_DETAIL PASSING T_PART_NUMBER
.
.
.
```

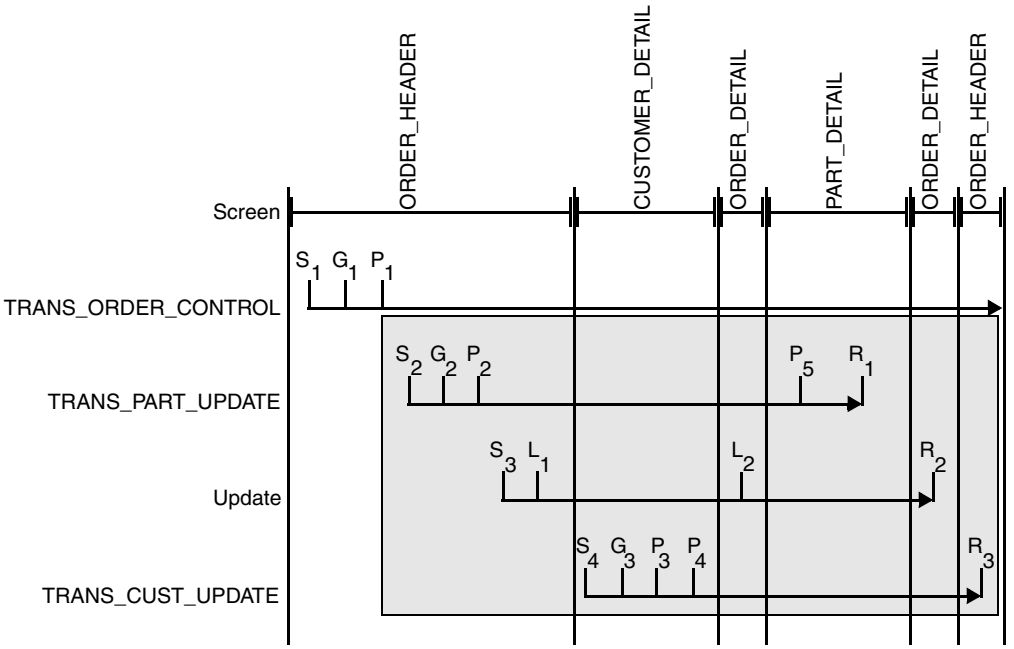
The PART_DETAIL screen is used for adding or updating part information. No commits are done on this screen. It uses the TRANS_PART_UPDATE transaction so that updates to part information are done in isolation from updates to the order information.

```
> SCREEN PART_DETAIL &
>        RECEIVING T_PART_NUMBER &
>        NOCOMMIT &
>        TRANSACTION MODEL CONCURRENCY
>
> TRANSACTION TRANS_PART_UPDATE INHERITED
> TEMPORARY T_PART_NUMBER NUMERIC
> FILE PART_DETAIL IN PERSONNEL PRIMARY &
>        TRANSACTION TRANS_PART_UPDATE FOR PROCESS, UPDATE
>
> FIELD PART_NUMBER LOOKUP NOTON PART_DETAIL
.
.
.
> PROCEDURE PREENTRY
>   BEGIN
>     LET PART_NUMBER OF PART_DETAIL = &
>         T_PART_NUMBER
```

```
>   END
```

The following is a series of steps that the user might perform and the actions QUICK takes in response.

	You do the following:		QUICK does the following:
1.	Start data entry by going into Entry mode on the ORDER_HEADER screen.	**1.**	The transaction, TRANS_ORDER_CONTROL is started (S_1) and is used to retrieve (G_1) and update ORDER_CONTROL (P_1). The transaction TRANS_PART_UPDATE is started (S_2) and is used to retrieve (G_2) and update PART_CONTROL (P_2).
2.	Enter the customer number and the lookup fails.	**2.**	The Update transaction is started (S_3) and is used for the lookup of CUSTOMER_NUMBER on CUSTOMER_DETAIL (L_1).
3.	Call the CUSTOMER_DETAIL screen to enter new customer information.	**3.**	The CUSTOMER_DETAIL screen is run. The transaction TRANS_CUST_UPDATE is started (S_4) and is used to get (G_3) and update CUSTOMER_CONTROL (P_3).
4.	Issue an Update Return command.	**4.**	The transaction TRANS_CUST_UPDATE is used to update CUSTOMER_DETAIL (P_4)
5.	Call ORDER_DETAIL from ORDER_HEADER.	**5.**	The ORDER_DETAIL screen is run
6.	On the ORDER_DETAIL screen, the lookup fails for a part number.	**6.**	The lookup is done on the Update transaction (L_2).
7.	Call PART_DETAIL to enter a new part number and issue an update command.	**7.**	An attempt to update PART_DETAIL record is made using TRANS_PART_UPDATE (P_5). A database constraint failure occurs. QUICK issues an error message and goes into Rollback Pending state.
8.	Issue two Return commands to back out of the screen.	**8.**	QUICK does a rollback.



To roll back, QUICK does the following:

1. Since the transaction, TRANS_PART_UPDATE, is locally active, it is rolled back (R_1). The earliest start point is S_2 and the rolled back transaction was used on an ancestor screen (ORDER_HEADER) for a get (G_2) and update (P_2).
2. Transaction TRANS_CUST_UPDATE was used for database operations after the earliest start point, S_2, to do get (G_3) and the two updates (P_3 and P_4).

   Transaction Update was used after S_2 on the ORDER_HEADER screen to do the lookup (L_1) and on the ORDER_DETAIL screen to do the lookup (L_2).

   Transaction TRANS_ORDER_CONTROL is active but not used for database operations since S_2.

   Consequently, the rollback is not local to the current screen.
3. If the Query transaction was active, it would be rolled back. The data buffers for PART_DETAIL screen are cleared and QUICK returns to ORDER_DETAIL.
4. Since the Update transaction was used to do a database operation (lookup L_2) on a local record, QUICK marks the transaction as locally active and rolls it back (R_2).

   As the Update transaction was started on an ancestor screen, the rollback is not local to the current screen. The data buffers for the ORDER_DETAIL screen are cleared and QUICK returns to ORDER_HEADER screen.
5. At this point, the highest-level screen involved in the rollback has been reached. QUICK rolls back the TRANS_CUST_UPDATE (R_3) since it was active in a subscreen.

   The data buffers for PART_CONTROL and ORDER_HEADER are cleared, and a rollback complete message is issued.

   The transaction TRANS_ORDER_CONTROL is still active and the data buffer for ORDER_CONTROL remains untouched.

The conceptual transaction consisted of the following transactions: TRANS_PART_UPDATE, TRANS_CUST_UPDATE, and Update.

Although this case illustrates how cascading rollbacks occur, it is not a good example of how to design a screen system. Good design should, where appropriate, ensure related operations are grouped under one transaction and distinct operations on other transactions.

As well, each time a new control number is required, the designer should use a COMMIT verb after the PUT verb to complete the operation and ensure that the control files are available to other users. The control number should also be calculated on the screen required. The calculation of the part control number should be moved to the PREENTRY procedure of the PART_DETAIL screen and each PREENTRY procedure should have a COMMIT verb at the end to complete the operation.

Also, customer and part information can be entered apart from the order entry. A separate Update transaction will achieve this, however, these transactions can be committed at update so that they are not rolled back as part of an unrelated cascading rollback. The NOCOMMIT option on the SCREEN statement for the PART_DETAIL and CUSTOMER_DETAIL screens should be removed.

Case 5: Rollback Pending Coexisting with Rollback Keep Buffers

In this case study, two read/write transactions are being used on the screen. One starts within the Update phase, but the other transaction could start either within the Update phase or prior to the Update phase.

If the Rollback Clear parameter in QKGO is set to N and the following QUICK screen is executed, there are times when one transaction can be rolled back immediately but the screen can be left in a Rollback Pending state.

In this example, it is assumed that a database is being used that requires two transactions for the Concurrency model.

```
>   SCREEN EMP_MAINT &
>      TRANSACTION MODEL CONCURRENCY
>
>   TRANSACTION WAGE_TXN READ WRITE
>
>   FILE EMPLOYEES IN  PERSONNEL PRIMARY
>   FILE WAGES IN PERSONNEL DESIGNER &
>      TRANSACTION WAGE_TXN FOR QUERY,  &
>      PROCESS, UPDATE
>   .
>   .
>   .
>   PROCEDURE DESIGNER WAGE
>      BEGIN
>         WHILE RETRIEVING WAGES VIA EMPLOYEE_NO &
>            USING EMPLOYEE_NO OF EMPLOYEES
>         ...
>      END
>   .
>   .
>   .
>   PROCEDURE UPDATE
>      BEGIN
>         PUT EMPLOYEES
>         PUT WAGES
>      END
>   BUILD
```

If the user finds an EMPLOYEES record, this will start the default read-only Query transaction. If the user then executes the designer procedure WAGES, this will start the designer transaction called WAGE_TXN. If the user then chooses to perform an Update, this will start the default read/write Update transaction in order to perform the PUT to EMPLOYEES. The PUT to WAGES will be executed as a part of the WAGE_TXN transaction.

In this example, the behavior in the event of an error during the Update phase depends on whether the error occurs on the PUT to EMPLOYEES or on the PUT to WAGES, and on whether the DESIGNER procedure was executed before the Update phase began.

If an error occurs during the PUT to WAGES, QUICK will attempt to roll back all locally active read/write transactions. Since the transaction WAGE_TXN started in a designer procedure, which is part of the Process Phase of the screen, QUICK cannot immediately roll back that transaction while keeping screen buffers. Since both the WAGE_TXN and UPDATE transaction are used in the Update phase of the screen, QUICK treats them as a single conceptual transaction and will place the screen in Rollback Pending state. If the user backs out at this point, both the UPDATE and WAGE_TXN transactions will be rolled back and the user will receive a message to that effect.

If, however, an error occurs while writing the EMPLOYEES record, QUICK will again attempt to roll back all locally active read/write transactions.

Since the WAGE_TXN was not used during the Update phase, QUICK can immediately roll back the Update transaction while keeping the data buffers intact on the screen. The WAGE_TXN cannot be rolled back immediately as it started before the Update phase.

The screen is therefore in Rollback Pending state and if the user exits the screen without correcting the error, the WAGE_TXN will be rolled back.

This difference in behavior can be attributed to the fact that the WAGE_TXN transaction was not used during the Update phase and therefore did not have to be considered by QUICK when determining whether Rollback Keep Buffers behavior was possible.

If the user never calls the WAGES designer procedure and simply executes the UPDATE procedure, both the WAGE_TXN and UPDATE transaction can be immediately rolled back while keeping data on the screen if an error is encountered while executing the PUT to WAGES.

Chapter 3: Relational Support in QTP

Overview

This chapter provides an overview of PowerHouse support for relational databases that are attached to your dictionary. You'll find information about

- QTP transaction models
- overriding the transaction defaults in QTP
- attaches and transactions in QTP
- tuning attaches in PowerHouse
- transaction error handling in QTP

QTP Transaction Model Overview

PowerHouse provides a default set of high-level transaction models that make it easier to code your application. With these models, you need not specify all the transactions and transaction control required for every PowerHouse application. PowerHouse establishes default transaction attributes and timing, associates activities with transactions, and controls transaction commits and rollback. In addition, the PowerHouse processing models incorporate built-in checking for such things as update conflicts, optimistic locking, and error recovery.

When the defaults aren't sufficient, PowerHouse provides options that allow you to customize, augment, or even replace the default processing at whatever level necessary, without giving up built-in support.

QTP Processing Environment

The QTP Transaction Models are:

- Consistency
- Concurrency

Since QTP is used primarily for single-user access where high data consistency is required, the default transaction model is Consistency. When you need high concurrent access, you can use QTP's second model, Concurrency.

Transaction Models in QTP

In QTP, the transaction model controls the number and type of transactions used in a request. Unlike QUICK, you can't explicitly choose which transaction model to use, instead you control this indirectly through the COMMIT AT statement.

Commit Frequency in QTP

The options on the COMMIT AT statement determine the transaction model and frequency of commits.

	Consistency Model	Concurrency Model
Commit Frequency	REQUEST RUN	FINAL INITIAL TRANSACTIONS sort-item START OF UPDATE

For more information about the COMMIT AT statement, see Chapter 3, "QTP Statements", in the *QTP Reference* book.

The Consistency Model in QTP

By default, QTP uses a read/write Consistency transaction for all database activity. The Consistency model provides high data consistency, at the cost of reduced concurrency.

Predefined Transactions

In this model, there is a one predefined transaction, the Consistency transaction. The Consistency transaction is a read/write transaction with a high isolation level and is used for all application activities. The database performs checking for conflicts among concurrent users' updates.

As a result of the high isolation level used in this model, each user's data is protected from being changed by other users (though not necessarily guaranteeing that a user will be able to update). As a potential side effect of enforcing this level of isolation, database products may protect (lock) more than just the data that has been "touched" by the user; this may diminish other users' ability to access data concurrently.

Creating Distinct Transactions

The number of Consistency transactions is controlled by the OPEN option of the SET FILE statement. For each unique open number, a separate Consistency transaction is used. For example, for open number 0 (the default), the transaction Consistency is used. For open number 1, the transaction Consistency_1 is used; for open number 2, Consistency_2 is used, and so on.

QTP does not look in the dictionary for a transaction with an appended open number.

The Consistency transaction may be read-only if QTP determines that no updates are required on that transaction. This can occur when the contents of a relation in a database are written to a subfile, or when SET FILE OPEN READ is used.

Using the Consistency Model with Sybase

In the Consistency model, when a table is used in both the input and output phases of the same QTP request, PowerHouse tries to prevent the data from being updated by another user prior to the output phase. For Sybase, this is accomplished using Sybase's HOLDLOCK option for each table that will be updated.

PowerHouse puts all requests that use the HOLDLOCK option into the same dbprocess by default. This is done in order to avoid deadlocks between Sybase dbprocesses. However, with Sybase, it is not possible to have more than one cursor open at any one time within a single dbprocess. (In order to process data from multiple cursors, either multiple concurrent dbprocesses must be used (one for each open cursor), or all the rows from the first cursor must be processed before the second cursor can be opened.)

These characteristics of Sybase affect the way that a designer can use the Consistency model in PowerHouse. For example, in QTP with Sybase, it is certainly possible to have multiple input and output tables in a request; however, it is not possible to have more than one table that is used in both the input and output phases with the default Consistency model.

As an example, the following does not cause a problem because only table A is referenced in both the input and output phases:

```
> ACCESS A LINK TO B LINK TO C
> OUTPUT A ...
> OUTPUT X ...
> OUTPUT Y ...
```

In the following example, however, both tables A and B are referenced for both input and output:

```
> ACCESS A LINK TO B LINK TO C
> OUTPUT A ...
> OUTPUT B ...
```

If, as shown in the previous example, a request has more than one input table and more than one of these tables is also used for output, you may get the error message:

```
"An internal system failure has occurred during operation 'asynchronous open'.
Attempt to initiate a new SQL Server operation with results pending."
```

Note: This error does not occur if individual unique input records are selected, such as by using CHOOSE with an explicit list of non-generic/non-range values that identify unique database records.

UNIX, Windows Examples

The following examples describe a few alternatives for avoiding or resolving situations where a request has more than one input table and more than one of these tables is also used for output.

Note: If any table is referenced more than once (e.g., using ALIASes) within the input or output phase of a request, you must be careful that you do not cause a deadlock between processes that are both trying to update the same database tables.

Alternative One

Use separate logical transactions for each of the tables that is used for both input and output.

In QTP, separate transactions can be created by using the SET FILE <file> OPEN <n> statement. A separate transaction is created for each distinct OPEN number used. The default transaction number is 0.

For example, if tables A,B, and C are used for input and output in a QTP request, then use the SET FILE statement to override the defaults so that each of the tables has a distinct OPEN number.

```
> ACCESS A LINK TO B LINK TO C
> OUTPUT A ...
> OUTPUT B ...
> OUTPUT C ...
> SET FILE B OPEN 1
> SET FILE C OPEN 2
```

For more information, see the SET statement in Chapter 3, "QTP Statements", in the *QTP Reference* book.

Alternative Two

Use SQL to create a single cursor for input.

In this case, PowerHouse issues the "select" for the cursor to Sybase, reads all the qualified records, then applies the updates to the appropriate output tables. By default, the data retrieved in the input phase will have shared locks applied that are released as soon as possible. This means that the input tables may not be protected from changes by other users prior to the output phase.

For example,

```
> SQL IN Base1 DECLARE AB_DATA CURSOR FOR &
>       SELECT A.COL1, A.COL2,B.COL1, B.COLN   &
```

```
>       FROM A, B WHERE A.COL1 = B.COL1
> ACCESS AB_DATA
> OUTPUT A IN Base1 UPDATE ...
> OUTPUT B IN Base1 UPDATE ...
```

Alternative Three

Assign ALIASes to the tables in the output phase.

In this case, PowerHouse considers the input tables and the output tables to be distinct, and therefore does not attempt to lock the input tables. As a result, each input table can be assigned to a separate dbprocess.

For example,

```
> ACCESS A LINK TO B
> OUTPUT A ALIAS A1 ...
> OUTPUT B ALIAS B1 ...
```

Note that the input tables will not be protected from change by other users prior to the output phase.

Alternative Four

Use direct SQL to update the table that is used for both input and output in QTP.

In some cases, it may be possible to replace QTP ACCESS and OUTPUT statements with direct SQL statements.

For example, instead of specifying ACCESS and OUTPUT statements, you could specify

```
> SQL IN Base1 UPDATE A SET COL1 = ... WHERE ...
> SQL IN Base1 UPDATE B SET COL1 = ... WHERE ...
```

Note that it may not be possible to replace all ACCESS and OUTPUT statements with direct SQL.

Alternative Five

Use the Concurrency model.

In this case, separate transactions are used for input and output in the Concurrency model. The input cursors do not use the HOLDLOCK option, therefore PowerHouse uses separate dbprocesses for each input cursor. The Concurrency model in QTP is invoked using either the COMMIT AT <commit point> statement, where the <commit point> is not RUN or REQUEST, or by using the SET LOCK FILE UPDATE statement. (Use of the COMMIT AT statement is the recommended method in 7.23.) Note, however, that the Concurrency model has higher transaction overhead (transactions are started and committed more frequently) and also does not protect input data from being changed by other users.

Remember, if any table is referenced more than once (e.g., using ALIASes) within the input or output phase of a request, you must be careful that you do not cause a deadlock between processes that are both trying to update the same database tables.

The Concurrency Model in QTP

The Concurrency model provides multi-user access to data and full functionality, yet still enforces a fairly high level of consistency. In this model, multiple users can read data, but only one user can update the same record at a time. When a user updates, PowerHouse verifies that there is no conflict among concurrent updates by re-reading the record and comparing the checksum of the record to the checksum of the original record. The record is updated if the checksums are equal. This approach generally results in high concurrency, since data is not locked until a user updates. This model is suitable for applications in which there are few "natural" update conflicts between users, or applications that mix data from relational and non-relational sources.

The checksum calculation omits:

- calculated columns. If they were included, the values could have been changed by the database, resulting in a checksum mismatch. This can easily occur if the user does multiple updates to the same row. Removing calculated columns from the checksum calculation eliminates these false errors.

- columns referenced by an ITEM statement with the OMIT option. The OMIT option specifically tells QTP to exclude the column in any updates typically because it is a read-only column or a calculated column. These columns are also excluded from the checksum.
- blob columns. These are excluded from the checksum calculation for performance reasons, as they can be very large.
- relational columns not referenced by the request. These are excluded because the checksum is based on the underlying SQL generated for the QTP request.

Predefined Transactions

In the Concurrency model, PowerHouse defines two predefined transactions:

- Query Transaction
- Update Transaction

Using the Concurrency Model for DB2, Sybase

QTP uses the Query transaction for retrieving and accessing records in the input phase, and the Update transaction to update records in the output phase.

During the output phase, each record that is to be updated is re-read on the Update transaction and compared to the record originally read on the Query transaction. If no change has occurred, the record is updated. Otherwise, an error message is issued and the update fails.

Using the Concurrency Model for ODBC

When PowerHouse connects to an ODBC data source, it queries the data source capabilities and tailors its behavior to that data source. Each relational database system has different capabilities for attaches and transactions.

Using the Concurrency Model for Oracle, Microsoft SQL Server, and ALLBASE/SQL

All ALLBASE/SQL, Microsoft SQL Server, and Oracle activities are associated with a single Update transaction.

The Update transaction starts as soon as access to the database is required, and ends when data is committed.

When updating in QTP, the records to be updated are locked, re-fetched, and checksummed to ensure that they have not been changed before being updated.

By default, PowerHouse uses the ALLBASE/SQL KEEP CURSOR option for Primary and Detail files to allow updating along a chain. This allows PowerHouse to retain a chain beyond a commit. In some cases, such as ordered retrieval, ALLBASE/SQL does not allow its KEEP CURSOR option to be used, and PowerHouse cannot retain the chain after the transaction has been committed.

Cursor Retention

By default, in the Concurrency Model, PowerHouse retains the read chain after a commit for all supported databases.

When PowerHouse connects to an ODBC data source, it queries the data source capabilities and tailors its behavior to that data source. For example, some ODBC data sources do not support cursor retention. PowerHouse determines if the ODBC data source supports cursor retention and uses it accordingly. Microsoft SQL Server supports cursor retention and PowerHouse uses it.

Creating Distinct Transactions

The number of Query transactions is controlled by the OPEN option of the SET FILE statement. For each unique open number, a separate Query transaction is used. For example, for open number 0 (the default), the transaction Query is used. For open number 1, the transaction Query_1 is used; for open number 2, Query_2 is used, and so on. The Query transaction is committed at the end of each request.

QTP does not look in the dictionary for a transaction with an appended open number.

There is only one Update transaction. The Update transaction is committed as specified on the COMMIT AT statement.

Transaction Attributes in QTP

Transaction attributes in QTP are controlled indirectly through the COMMIT AT and SET FILE statement.

The behavior of the transaction model depends on the attributes that are supported by QTP and the target database. This section summarizes and compares some of the transaction attributes supported by QTP and the supported databases.

Default Transaction Attributes in QTP

The default attributes for the QTP transactions are:

Transaction	Model	Access	Isolation Level
Consistency	Consistency	read/write	SERIALIZABLE
Query	Concurrency	read-only	READ COMMITTED
Update	Concurrency	read/write	REPEATABLE READ

Transaction Access Types

The transaction access type determines the type of activities that can be performed by a transaction and the type of transaction started in the associated database. The Query transaction is read-only. The Update and Consistency transactions are read/write, unless SET FILE OPEN READ is used, in which case they are read-only.

Isolation Levels

Isolation levels specify the degree to which each transaction is isolated from the actions of other transactions. Different database products support different transaction isolation levels; some offer a choice of isolation levels, some provide just one. Low levels of isolation mean that transactions are not well protected from each other; in other words, simultaneous transactions may get inconsistent results. Higher levels of isolation generally mean that transactions are better protected from each other. At the highest levels, each transaction may be entirely unaware of changes being made by other transactions.

Lower isolation levels generally allow higher concurrency with a potential loss of consistency, while higher isolation levels provide high consistency but generally result in lower concurrency.

The support available for the various isolation level options offered in QTP depends on the support provided by the underlying database software.

If a database doesn't support a specified isolation level, PowerHouse uses the next available higher isolation level. If a higher level is unavailable, PowerHouse uses the highest available lower level.

When isolation levels are upgraded or downgraded

- for user defined transactions, PowerHouse issues a warning message at compile-time
- for the default PowerHouse transactions or for inherited transactions, PowerHouse does not issue warning messages

The following are some of the terms used to describe isolation levels. The levels are listed from lowest to highest, although the levels are not strictly incremental:

Isolation Level	Description
READ UNCOMMITTED	Allows a transaction to see all changes made by other transactions, whether committed or not. Also known as a "dirty read".
READ COMMITTED	Allows a transaction to read any data that has been committed by any transaction as of the time the read is done.
STABLE CURSOR	Indicates that while a transaction has addressability to a record (that is, has just fetched it), no other transaction is allowed to change or delete it.
REPEATABLE READ	Allows any data that has been read during a transaction to be re-read at any point within that transaction with identical results.
PHANTOM PROTECTION	Doesn't allow a transaction to see new records, or "phantoms", that did not exist when the transaction started.
SERIALIZABLE	Indicates that the results of the execution of a group of concurrent transactions must be the same as would be achieved if those same transactions were executed serially in some order.

Database-Specific Transaction Attributes

This table shows the expected behavior when using different database transaction attributes. PowerHouse issues a warning message when you specify options for features that are not supported by the target database.

ODBC: When PowerHouse connects to an ODBC data source, it queries the data source capabilities and tailors its behavior to that data source. One aspect that is determined is the transaction isolation levels that the data source supports.

Transaction Attribute	ALLBASE /SQL	DB2	Microsoft SQL Server	Oracle	Oracle Rdb	Sybase
Isolation levels						
READ UNCOMMITTED	READ UNCOMMITTED	READ UNCOMMITTED (DB2's Uncommitted Read)	READ UNCOMMITTED	READ COMMITTED[2]	READ COMMITTED	Warning[1]
READ COMMITTED	READ COMMITTED	READ COMMITTED (DB2's Cursor Stability)	READ COMMITTED	READ COMMITTED[2]	READ COMMITTED	Warning[1]
STABLE CURSOR	STABLE CURSOR	REPEATABLE READ (DB2's Read Stability)	REPEATABLE READ	READ COMMITTED and locks fetched rows[3]	REPEATABLE READ	Warning[1]

Transaction Attribute	ALLBASE /SQL	DB2	Microsoft SQL Server	Oracle	Oracle Rdb	Sybase
REPEATABLE READ	REPEATABLE READ	REPEATABLE READ (DB2's Read Stability)	REPEATABLE READ	for Read Only transactions - uses Oracle's READ ONLY for Read/Write transaction - uses READ COMMITTED and locks fetched rows	REPEATABLE READ	Warning[1]
PHANTOM PROTECTION	REPEATABLE READ	SERIALIZABLE (DB2's Repeatable Read)	SERIALIZABLE	SERIALIZABLE	SERIALIZABLE	Warning[1]
SERIALIZABLE	REPEATABLE READ	SERIALIZABLE (DB2's Repeatable Read)	SERIALIZABLE	SERIALIZABLE	SERIALIZABLE	Warning[1]
Deferring Constraints	Supported	Warning	Warning	Warning	Warning	Warning
Transaction Priority	Supported	Warning	Warning	Warning	Warning	Warning
Reserving List	Supported[4]	Warning	Warning	Supported	Supported	Warning
Wait for locked database resources	Supported	Warning	[NO]DBWAIT	Supported	Supported	Warning
Read-only Read/write options	Supported	Warning	n/a	Supported	Supported	Supported

Supported: supported both by PowerHouse and the target database.

Warning: a warning message results. The transaction attribute is supported by PowerHouse but not by the target database.

[1] *All locks and lock escalation are managed by Sybase and cannot be overridden.*

[2] *Uses Oracle statement-level read consistency for read/write transactions.*

[3] *Uses Oracle transaction-level read consistency for read-only transactions, and statement-level read consistency for read/write transactions.*

[4] *PowerHouse issues lock table requests for tables in the reserving list.*

Isolation Levels and Generated SQL Limitation in Oracle

Since Oracle does not support explicit control of transaction isolation levels, PowerHouse adds a FOR UPDATE clause to all SQL cursor specifications sent to an Oracle database whenever a PowerHouse transaction requests an isolation level of Repeatable Read or higher.

By default, transactions used in the Consistency model in QUICK and QTP have this characteristic, as would any designer-defined transactions with an isolation level of Repeatable Read, Phantom Protection, or Serializable.

As a result of Oracle restrictions on the use of the FOR UPDATE clause, the SQL generated by PowerHouse may result in an invalid Oracle specification. For example, the FOR UPDATE clause is not allowed in Oracle if the query includes DISTINCT, GROUP BY, any set operator, or any group function. Using any of these features in a cursor will result in an error message being returned from Oracle, indicating that FOR UPDATE is not allowed in the statement.

To avoid this situation, ensure that these requests are executed within a transaction with a low isolation level. The default Concurrency or Optimistic models use low isolation levels by default. For further information about setting and overriding transaction isolation levels, please refer to the TRANSACTION statement in Chapter 3, "QTP Statements", in the *QTP Reference*, or to the same statement in Chapter 3, "QDESIGN Statements", in the *QDESIGN Reference*.

Consult your Oracle database documentation for more information about restrictions on the use of the FOR UPDATE clause.

The Consistency Model and Oracle Error ORA-08177

PowerHouse 8.4x and up supports the Serializable isolation level within Oracle versions 8i and above. Oracle generates an error when a serializable transaction tries to update or delete data modified by a transaction that commits after the serializable transaction began:

```
ORA-08177: Cannot serialize access for this transaction
```

Since QTP defaults to the Consistency model, which in turn defaults to the Serializable isolation level, we recommend Serializable isolation be only used for environments:

- with large databases and short transactions that only update a few rows
- where there is a relatively low chance that two concurrent transactions will modify the same rows
- where relatively long-running transactions are primarily read-only.

For other QTP environments, we recommend that you set Consistency transactions to REPEATABLE READ in the PDL dictionary. This can be done with the TRANSACTION statement:

```
TRANSACTION QTP_CONSISTENCY REPEATABLE READ ;for QTP only
```

or

```
TRANSACTION CONSISTENCY REPEATABLE READ ;for both QTP and QDESIGN
```

PowerHouse sets a REPEATABLE READ transaction as an Oracle READ COMMITTED transaction with the FOR UPDATE clause, as was done in versions of PowerHouse prior to 8.4x.

Some customers have observed the Oracle error ORA-081777 with a single user updating over 1000 rows. A workaround for this is to set a higher value of the INITRANS parameter for those tables. Please see the Oracle reference documentation for details.

Locking Strategy

QTP locks tables that it reads or updates. The type of lock is determined by the FILE OPEN option on the SET statement, as specified in the following table:

QTP Option	ALLBASE/SQL	Oracle	Oracle Rdb	Sybase
EXCLUSIVE	Exclusive	Exclusive Table Lock	Exclusive Table Lock	not applicable
SHARE	Shared	Row Exclusive Table Lock	Shared Table Lock	not applicable

DB2, ODBC, Microsoft SQL Server: The access and exclusivity options of the SET FILE OPEN statement are not supported by DB2 or ODBC (and hence Microsoft SQL Server).

How Reserving Works in QTP

PowerHouse supports reserving if explicit locks are supported by the database. If the explicit lock feature is supported, PowerHouse attempts to lock each table in the reserving list using an explicit lock request. Internally, this is done by generating a SHARED READ reserving table for each table in the ACCESS statement and a SHARED WRITE reserving table for each table in an OUTPUT statement.

The reserving option is supported by **Oracle, Oracle Rdb,** and **ALLBASE/SQL**. Reserving is not supported by **ODBC, Sybase, DB2** or **Microsoft SQL Server.**

The RESERVING FOR option of the dictionary TRANSACTION statement is ignored by QTP. However, if reserving is supported by the database, QTP automatically locks the tables that are referenced in QTP source code.

Overriding the Transaction Defaults in QTP

The following tables summarize the statements and program parameters that you can use to override transaction defaults, such as, specifying which transaction model to use and the commit frequency.

QTP Statements	Options	Purpose/Effect
COMMIT AT	REQUEST RUN	QTP uses the Consistency model.
	FINAL INITIAL TRANSACTIONS sort-item START OF UPDATE	QTP uses the Concurrency model.

Program Parameters	Purpose/Effect
dbwait|nodbwait	Determines what happens when a requested resource is in use.

PDL Statement	Purpose/Effect
TRANSACTION	Used to override the default transaction characteristics.

The behavior of the transaction that QTP uses may be customized. In QTP, the attributes for the Query, Consistency, and Update transactions are determined as follows:

1. QTP sets the attributes by looking in the dictionary for a transaction named QTP_QUERY.
2. If a QTP_QUERY transaction has not been defined in PDL, then QTP sets the attributes by looking in the dictionary for a transaction named QUERY.
3. If there is no QTP_QUERY or QUERY transaction defined in the dictionary, then the transaction name defaults to QUERY. Its attributes are set to the default values specified for the options of the TRANSACTION statement in PDL.

   The same three-step process applies for determining attributes for the Consistency and Update transactions, in which case QTP looks for the QTP_CONSISTENCY, CONSISTENCY, QTP_UPDATE, and UPDATE transactions.

QTP does not look in the dictionary for a transaction with an appended open number. For example, QTP will not look in the dictionary for a transaction named QUERY_02.

Attaches and Transactions in QTP

PowerHouse manages attaches and transactions to access relational database systems. An attach opens the database and makes the PowerHouse application known to the database. A transaction is used to access the database. All requests to read, insert, update, or delete database information are done by associating the requests with a transaction.

Each relational database system has different capabilities for attaches and transactions.

The following table outlines the different requirements for the supported databases:

Database	Requirement
ALLBASE/SQL	Requires a separate attach for each distinct transaction.
DB2	Does not require a separate attach for each transaction.
Microsoft SQL Server	Does not require a separate attach for each transaction.
ODBC	When PowerHouse connects to an ODBC data source it queries the data source capabilities and tailors its behavior to that data source. Each relational database system has different capabilities for attaches and transactions.
Oracle	Requires a separate attach for each distinct transaction.
Oracle Rdb	Does not require a separate attach for each transaction.
Sybase	PowerHouse associates activities with separate Sybase dbprocesses. A single PowerHouse transaction may map to multiple dbprocesses, since a single dbprocess cannot process more than one type of request at a time.

Recycling Attaches

As attaches consume resources, PowerHouse tries to minimize the number of attaches it uses. When a transaction ends by either being committed or rolled back, instead of issuing a detach call, PowerHouse preserves the attach for future use. PowerHouse re-uses an attach to start a new transaction when another attach is needed and the attach is for the right database. A new attach is issued if there are no attaches available or none match.

QTP compiles all of the requests in a run and can optimize database attaches across different QTP requests to minimize the number of attaches required. The number of attaches and the number of database transactions varies depending on the commit timing specified.

Consider the following run where the EMPLOYEES and BILLINGS tables are in databases Base1 and Base2 respectively, and the PROJECTS table is in database Base3. In this example Base1 and Base2 are both of the same database type, such as Sybase and Base3 is a different database type.

```
RUN BATCH_DELETE

REQUEST BILLINGS_DELETE
ACCESS EMPLOYEES IN Base1
CHOOSE EMPLOYEE PARM
OUTPUT BILLINGS IN Base2 DELETE

REQUEST PROJECTS_DELETE
ACCESS EMPLOYEES IN Base1
CHOOSE EMPLOYEE PARM
OUTPUT PROJECTS IN Base3 DELETE
```

The Consistency Model

Commit at Run

By default, the Consistency PowerHouse transaction is committed at the end of the run. The transaction is used to access and update the tables in the previous example. When this run is executed, QTP:

- Starts database transactions that are attached to Base1 and Base2. The attach to Base1 is done to retrieve rows from the EMPLOYEES table; the attach to Base2 deletes the appropriate rows in the BILLINGS table.
- Starts another database transaction attached to Base3 to delete the appropriate rows in the PROJECTS table.

Commit At Request

If a COMMIT AT REQUEST statement is specified for this run, when executing the BILLINGS_DELETE request, as part of the PowerHouse Consistency transaction, QTP attaches to both Base1 and Base2 databases. When the request completes, the PowerHouse and database transactions are committed, but the attaches remain active.

However, when the PROJECTS_DELETE request is executed, a new Consistency transaction starts which then starts a database transaction to access the EMPLOYEES table, and re-uses the attach started in the first request. Since the PROJECTS table is in a different database type than the unused attaches, a different database attach to delete the PROJECTS table is required.

The Concurrency Model

For the BILLINGS_DELETE request, to retrieve the EMPLOYEES table, QTP starts the Query transaction which in turn starts a database transaction attached to Base1 to retrieve the EMPLOYEES table. To delete rows in the BILLINGS table, QTP starts the PowerHouse Update transaction which requires a database transaction attached to Base2.

If either of the two PowerHouse transactions are still active when this request is completed, they are committed which, in turn, means that the underlying database transactions are committed. The two attaches used in this request are kept for future use.

In the PROJECTS_DELETE request, to retrieve the EMPLOYEES table, QTP starts the Query transaction which in turn starts a database transaction attached to Base1 to retrieve the EMPLOYEES table. The attach used in the first request to retrieve EMPLOYEES information is re-used for this transaction. To delete rows in the PROJECTS table, QTP starts the PowerHouse Update transaction which requires a database transaction attached to Base3.

In summary, in the previous example, the Concurrency model uses three attaches:

- an attach to Base1 used by the Base1 database transaction that is needed by the PowerHouse Query transaction in both QTP requests.
- an attach to Base2 used by the Base2 database transaction that is needed by the PowerHouse Update transaction in the first QTP request.
- an attach to Base3 used by the Base3 database transaction that is needed by the PowerHouse Update transaction in the second QTP request.

Transaction Error Handling in QTP

The ON ERRORS TERMINATE option on the following statements cause QTP to roll back active transactions:

- CHOOSE
- DEFINE
- OUTPUT
- REQUEST
- SUBFILE
- SQL DML statements

Therefore, you should ensure that the commit timing you specify with the COMMIT AT statement is in line with the ON ERRORS TERMINATE REQUEST or RUN option.

If you use COMMIT AT RUN and specify the ON ERRORS TERMINATE REQUEST option, QTP rolls back the current request as well as all the previous requests, and then proceeds to the next request.

Conversely, if you use COMMIT AT REQUEST and specify the ON ERRORS TERMINATE RUN option, QTP rolls back only the current request (because the previous ones are committed), and then terminates the run.

Index

D

E

Q

R

S

T

U

V

W