
Enterprise PeopleTools 8.51 PeopleBook: Application Engine

October 2011

Copyright © 1988, 2011, Oracle and/or its affiliates. All rights reserved.

Trademark Notice

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

License Restrictions Warranty/Consequential Damages Disclaimer

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

Warranty Disclaimer

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

Restricted Rights Notice

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

Hazardous Applications Notice

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Third Party Content, Products, and Services Disclaimer

This software and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third party content, products and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third party content, products or services.

Contents

Preface

Application Engine Preface	xi
Application Engine	xi
PeopleBooks and the PeopleSoft Online Library	xi

Chapter 1

Getting Started With Application Engine	1
Application Engine Overview	1
Application Engine Implementation	2

Chapter 2

Understanding Application Engine	5
Application Engine Fundamentals	5
Meta-SQL	5
Application Engine Program Elements	6
Sections	6
Steps	6
Actions	6
State Records	8
Application Engine Program Types	8
Application Engine Program Types	8
Daemon Program Type	9
Transform Program Type	10

Chapter 3

Creating Application Engine Programs	11
Viewing Application Engine Programs	11
Using Definition View	12
Using Program Flow View	13
Switching Between Definition and Program Flow Views	15

Using the Refresh Option	16
Filtering View Contents	16
Printing Program and Flow Definitions	19
Creating, Opening, and Renaming Programs	19
Creating New Programs	20
Opening Existing Programs	20
Renaming Programs	20
Copying or Moving Program Elements	21
Testing Application Engine Programs	21
Setting Program Properties	23
Accessing Properties	23
Setting General Properties	23
Setting State Record Properties	23
Specifying Temporary Tables	25
Setting Advanced Properties	26
Adding Sections	28
Understanding Sections	28
Inserting Sections	29
Locating Sections	29
Setting Section Properties	30
Adding Steps	31
Inserting Steps	31
Setting Up Step Properties	32
Specifying Actions	33
Understanding Actions	34
Inserting Actions	35
Setting Action Properties	35
Specifying SQL Actions	37
Specifying Do Actions	38
Specifying PeopleCode Actions	40
Specifying Call Section Actions	41
Specifying Log Message Actions	42
Specifying XSLT Actions	43

Chapter 4

Developing Efficient Programs	45
Using State Records	45
Understanding State Records	45
Sharing State Records	46
Choosing a Record Type for State Records	47
Setting Commits	47
Reusing Statements	48

Using Bulk Insert	50
Using Set Processing	50
Understanding Set Processing	51
Using Set Processing Effectively	51
Avoiding Row-by-Row Processing	53
Using Set Processing Examples	54

Chapter 5

Using Meta-SQL and PeopleCode	59
Understanding Application Engine Meta-SQL	59
Using PeopleCode in Application Engine Programs	60
Understanding PeopleCode and Application Engine Programs	60
Deciding When to Use PeopleCode	62
Considering the Program Environment	63
Accessing State Records with PeopleCode	64
Using If/Then Logic	65
Using PeopleCode in Loops	65
Using the ASection Class	66
Making Synchronous Online Calls to Application Engine Programs	66
Using the File Class	67
Calling COBOL Modules	67
Calling PeopleTools APIs	70
Using the CommitWork Function	71
Calling WINWORD Mail Merge	71
Using PeopleCode Examples	71
Including Dynamic SQL	74
Application Engine Meta-SQL Reference	74
%Abs	74
%AeProgram	75
%AeSection	75
%AeStep	75
%AsOfDate	75
%AsOfDateOvr	75
%BINARYSORT	76
%Bind	76
%Cast	78
%ClearCursor	79
%COALESCE	80
%Comma	80
%Concat	80
%CurrentDateIn	81
%CurrentDateOut	81

%CurrentDateTimeIn	81
%CurrentDateTimeOut	82
%CurrentTimeIn	82
%CurrentTimeOut	82
%DateAdd	83
%DateDiff	83
%DateIn	83
%DateNull	83
%DateOut	84
%DatePart	84
%DateTimeDiff	84
%DateTimeIn	85
%DateTimeNull	85
%DateTimeOut	85
%DecDiv	85
%DecMult	86
%DTTM	86
%EffDtCheck	86
%Execute	87
%ExecuteEdits	87
%FirstRows	89
%GetProgText	90
%InsertSelect	91
%InsertSelectWithLongs	91
%JobInstance	91
%Join	91
%LeftParen	92
%Like	92
%LikeExact	92
%List	93
%ListBind	95
%ListEqual	96
%Mod	97
%Next and %Previous	98
%NoUpperCase	98
%NumToChar	99
%ProcessInstance	99
%ResolveMetaSQL	99
%ReturnCode	101
%RightParen	102
%Round	102
%RoundCurrency	102
%RunControl	103
%Select	103
%SelectInit	104

%Space	105
%SQL	105
%SQLRows	106
%Substring	106
%Table	107
%Test	107
%TextIn	108
%TimeAdd	108
%TimeIn	108
%TimeNull	108
%TimeOut	109
%TimePart	109
%TrimSubstr	109
%Truncate	110
%TruncateTable	110
%UpdateStats	111
%Upper	115

Chapter 6

Managing Application Engine Programs	117
Running Application Engine Programs	117
Understanding Program Execution Options	117
Creating Process Definitions	118
Listing Process Definition Parameters	120
Starting Programs with the Application Engine Process Request Page	121
Using PeopleCode to Invoke Application Engine Programs	122
Using the Command Line to Invoke Application Engine Programs	123
Debugging Application Engine Programs	126
Enabling the Application Engine Debugger	126
Setting Debugging Options	127
Restarting Application Engine Programs	131
Understanding Restart	131
Determining When to Use Restart	132
Controlling Abnormal Terminations	134
Restarting Application Engine Programs	134
Starting Application Engine Programs from the Beginning	135
Enabling and Disabling Restart	136
Caching the Application Engine Server	136
Freeing Locked Temporary Tables	137

Chapter 7

Calling Application Engine Programs from COBOL	139
Adding Copybooks to COBOL Programs	139
Assigning Copybook Values	141
Handling COBOL Errors	145

Chapter 8

Tracing Application Engine Programs	147
Understanding Tracing Application Engine Programs	147
Understanding Trace Results	147
Trace File Sections	148
Step Trace	151
SQL Trace	151
Statement Timings Trace	151
Database Optimizer Trace	153
Enabling Application Engine Tracing	156
Setting Command Line Options	156
Setting Parameters in Server Configuration Files	157
Setting Options in PeopleSoft Configuration Manager	158
Locating Trace Files	158

Chapter 9

Using Temporary Tables	161
Understanding Temporary Tables	161
Creating Temporary Table Instances	163
Understanding Temporary Table Instances	163
Defining Temporary Tables	164
Setting the Number of Temporary Table Instances	164
Building Table Instances	166
Managing Temporary Table Instances	166
Understanding Temporary Table Instance Numbers	166
Assigning Temporary Tables to Programs	167
Adjusting Meta-SQL	170
Making External Calls	172
Viewing Temporary Table Usage	174
Viewing Temporary Table Usage by Record	174
Viewing Temporary Table Settings by Program	175

Viewing Online Instance Usage	176
Resolving the Temporary Table Usage Warning Message	176
Index	179

Application Engine Preface

This PeopleBook describes Application Engine.

Application Engine

Application Engine is designed to help you develop, test, and run background SQL processing programs. This PeopleBook explains the concepts and advantages of Application Engine, how to develop Application Engine programs in Application Designer, how to run and debug programs, and the use of the special tools to maintain your programs.

The "About These PeopleBooks Preface" contains general product line information, such as related documentation, common page elements, and typographical conventions.

Note. *DB2 UDB for OS/390 and z/OS* is the official IBM name for the database management system (DBMS). In the current PeopleTools release, Oracle no longer supports the OS/390 operating system, only z/OS, its replacement. For the sake of brevity, this PeopleBook sometimes refers to DB2 UDB for OS/390 and z/OS as *DB2 z/OS*, and it sometimes refers to DB2 UDB for Linux, UNIX, and Windows as *DB2 UNIX/NT*.

PeopleBooks and the PeopleSoft Online Library

A companion PeopleBook called *PeopleBooks and the PeopleSoft Online Library* contains general information, including:

- Understanding the PeopleSoft online library and related documentation.
- How to send PeopleSoft documentation comments and suggestions to Oracle.
- How to access hosted PeopleBooks, downloadable HTML PeopleBooks, and downloadable PDF PeopleBooks as well as documentation updates.
- Understanding PeopleBook structure.
- Typographical conventions and visual cues used in PeopleBooks.
- ISO country codes and currency codes.
- PeopleBooks that are common across multiple applications.
- Common elements used in PeopleBooks.
- Navigating the PeopleBooks interface and searching the PeopleSoft online library.
- Displaying and printing screen shots and graphics in PeopleBooks.
- How to manage the locally installed PeopleSoft online library, including web site folders.

- Understanding documentation integration and how to integrate customized documentation into the library.
- Application abbreviations found in application fields.

You can find *PeopleBooks and the PeopleSoft Online Library* in the online PeopleBooks Library for your PeopleTools release.

Chapter 1

Getting Started With Application Engine

This chapter provides an overview of Application Engine and discusses:

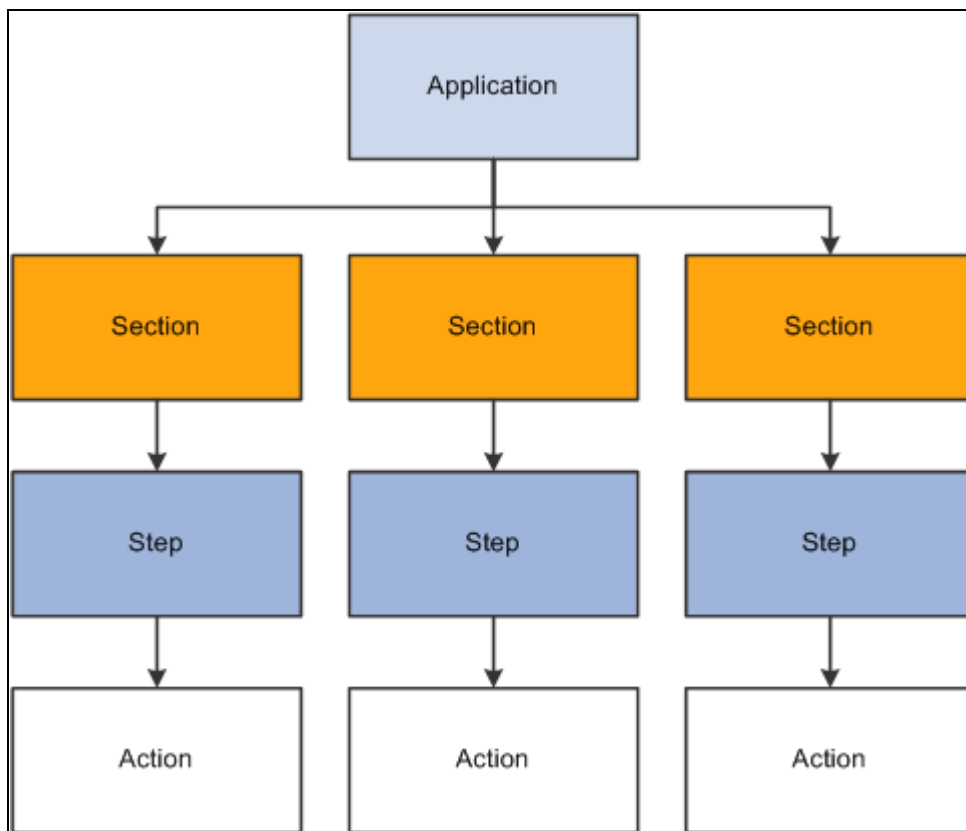
- Application Engine implementation
- Other sources of information

Application Engine Overview

Application Engine is a PeopleTool designed to help you develop background SQL processing programs. This tool is intended to be used by developers with knowledge of SQL, SQL tools, and PeopleTools.

Application Engine offers you an alternative to writing COBOL or SQR programs for background SQL processing. While Application Engine does not generate, parse, or understand SQL, it does execute SQL that you provide.

This diagram shows the program structure of Application Engine:



Application Engine Program Structure

Application Engine Implementation

This section provides information to consider before you begin to use Application Engine.

Implementation of Application Engine can be divided into the following activities:

- Set up properties.
- Specify actions.
- Create temporary table instances.
- Set up debugging options.
- Enable application engine tracing.

Setting Up Properties

To set up Application Engine properties, you perform the following steps:

Step	Reference
1. Set up program properties.	See Chapter 3, "Creating Application Engine Programs," Setting Program Properties, page 23.
2. Set up section properties.	See Chapter 3, "Creating Application Engine Programs," Setting Section Properties, page 30.
3. Set up step properties.	See Chapter 3, "Creating Application Engine Programs," Setting Up Step Properties, page 32.
4. Set up action properties.	See Chapter 3, "Creating Application Engine Programs," Setting Action Properties, page 35.

Specifying Actions

To modify the action properties, you perform the following steps:

Step	Reference
1. Specify SQL actions.	See Chapter 3, "Creating Application Engine Programs," Specifying SQL Actions, page 37.
2. Specify Do actions.	See Chapter 3, "Creating Application Engine Programs," Specifying Do Actions, page 38.
3. Specify PeopleCode actions.	See Chapter 3, "Creating Application Engine Programs," Specifying PeopleCode Actions, page 40.
4. Specify Call Section actions.	See Chapter 3, "Creating Application Engine Programs," Specifying Call Section Actions, page 41.
5. Specify Log Message actions.	See Chapter 3, "Creating Application Engine Programs," Specifying Log Message Actions, page 42.

Creating Temporary Table Instances

To set up temporary tables to improve performance, you perform the following steps:

Step	Reference
1. Define temporary tables.	See Chapter 9, "Using Temporary Tables," Defining Temporary Tables, page 164.
2. Set up the number of temporary table instances.	See Chapter 9, "Using Temporary Tables," Setting the Number of Temporary Table Instances, page 164.
3. Build table instances.	See Chapter 9, "Using Temporary Tables," Building Table Instances, page 166.

Setting Up Debugging Options

To set up debugging options for Application Engine programs, you perform the following steps:

Step	Reference
1. Enable the Application Engine debugger.	See Chapter 6, "Managing Application Engine Programs," Enabling the Application Engine Debugger, page 126.
2. Set up debugging options.	See Chapter 6, "Managing Application Engine Programs," Setting Debugging Options, page 127.

Enabling Application Engine Tracing

To trace Application Engine programs, you perform the following steps:

Step	Reference
1. Set command line options.	See Chapter 8, "Tracing Application Engine Programs," Setting Command Line Options, page 156.
2. Set parameters in server configuration files.	See Chapter 8, "Tracing Application Engine Programs," Setting Parameters in Server Configuration Files, page 157.
3. Set options in Configuration Manager.	See Chapter 8, "Tracing Application Engine Programs," Setting Options in PeopleSoft Configuration Manager, page 158.

Other Sources of Information

In addition to implementation considerations presented in this chapter, take advantage of all PeopleSoft sources of information, including the installation guides, release notes, and PeopleBooks.

See Also

["Application Engine Preface," page xi](#)

PeopleTools 8.51 PeopleBook: Getting Started with Enterprise PeopleTools

Chapter 2

Understanding Application Engine

You use Application Engine to develop batch or online programs that perform high-volume, background processing against your data.

This chapter discusses:

- Application Engine fundamentals.
- Meta-Structured Query Language (SQL).
- Application Engine program elements.
- Application Engine program types.

Application Engine Fundamentals

Application Engine comprises two distinct components—a designer where you define your batch program and the runtime environment where you run and monitor your program.

In Application Engine, a *program* is a set of SQL statements, PeopleCode, and program control actions that enable looping and conditional logic. A program is defined in Application Designer and performs a business process. You can use Application Engine for straight, row-by-row processing, but the most efficient Application Engine programs are written to perform set-based processing.

Application Engine does not generate SQL or PeopleCode. It executes the SQL and PeopleCode that you include in an Application Engine action as part of your program.

Application Engine is designed for batch processing where you have data that must be processed without user intervention—for example, calculating salaries in payroll processing (although not printing the checks). Another example might be converting money from one currency to another.

Meta-SQL

You can write SQL within Application Engine, or you can copy SQL statements into Application Engine from any SQL utility with few, if any, changes. This enables you to write and tune SQL statements before you try to incorporate them into an Application Engine program.

Database platforms can have different syntax rules, especially in regard to date, time, and other numeric calculations. Generally, you can work around syntax differences using PeopleSoft meta-SQL, which Application Engine supports. Meta-SQL is a set of predefined terms (meta-strings), designed to replace relational database management system (RDBMS)-specific SQL syntax with a common syntax.

In addition, PeopleSoft meta-SQL enables you to dynamically generate portions of SQL code. For example, to join two tables based on their common keys, use the following meta-string:

```
%Join(COMMON_KEYS, PSAESECTDEFN ABC, PSAESTEPDEFN XYZ )
```

At runtime, the function would be expanded into the following:

```
ABC.AE_APPLID = XYZ.AE_APPLID  
AND ABC.AE_SECTION = XYZ.AE_SECTION  
AND ABC.DBTYPE = XYZ.DBTYPE  
AND ABC.EFFDT = XYZ.EFFDT
```

Application Engine Program Elements

A Application Engine program comprises the set of processes to execute a given task. It is made up of these key elements:

- Sections
- Steps
- Actions
- State records

Sections

Sections comprise one or more steps and are equivalent to a COBOL paragraph or an SQR procedure. All Application Engine programs must contain at least one section entitled *MAIN*.

A section is a set of ordered steps that is executed as part of a program. You can call sections (and other programs) from steps within other sections.

A program must contain at least one section. The execution of the program always starts with the section defined as *MAIN*.

Steps

Steps are the smallest unit of work that can be committed within a program. Although you can use a step to execute a PeopleCode command or log a message, typically you use a step to execute a SQL statement or to call another section. The SQL or PeopleCode that a step executes are the actions within the step.

When a section is called, its steps execute sequentially. Every program begins by executing the first step of the required section called *MAIN* and ends after the last step in the last section completes successfully.

Actions

You can specify multiple types of actions for inclusion within a step. Multiple actions are commonly associated with a single step.

Do Actions

Do actions contain a SQL **Select** statement designed to return results on which subsequent actions depend. For instance, if a **Select** statement returns no rows, then subsequent actions may not need to execute. A Do action is equivalent to a COBOL **Perform** statement and has similar constructs.

The four types of Do actions are:

- Do While
- Do When
- Do Select
- Do Until

SQL

Most SQL actions contain a single SQL statement. These actions can perform the following types of SQL statements:

- **Update**
- **Delete**
- **Insert**
- **Select**

The SQL action differs from the Do actions, which also contain SQL, in that the SQL action does not control the flow of the program.

PeopleCode

You can include PeopleCode in the PeopleCode action. Application Engine PeopleCode provides an excellent way to build dynamic SQL, perform simple if/else edits, set defaults, and other operations that do not require a trip to the database. It also enables you to reference and change active Application Engine state records.

Most importantly, PeopleCode provides access to the PeopleSoft integration technologies, such as PeopleSoft Integration Broker, Component Interfaces, Business Interlinks, and file processing.

Log Message

You use a Log Message action to write a message to the message log based on a condition in your program. This gives your program multilanguage capability. The system stores the message generically as a message set, message number, and parameter values. When a user views the messages using the Application Engine Message Log page, the system retrieves the appropriate message string from the message catalog based on the user's language preference.

Call Section

You can also insert an action that calls another section. The called section can be in the same program as the calling section, or it can be in an external program. This enables you to chunk your program into more maintainable, reusable pieces. If a section already exists in one program, rather than copying it into another program, just call it.

Note. Application Engine supports up to 99 levels of nested Call Section actions. For example, the first called section can call a second, which can call a third, and so on, up to 99 calls.

State Records

A state record is a PeopleSoft record that must be created and maintained by the Application Engine developer. This record defines the fields a program uses to pass values from one action to another. Think of the fields of the Application Engine state record as the working storage for your Application Engine program.

An Application Engine state record can be either a physical record or a work record, and any number of state records can be associated with a program. Physical state records must be keyed by process instance.

Application Engine Program Types

This section discusses:

- Application Engine program types.
- Daemon program type.
- Transform program type.

Application Engine Program Types

Application Engine has five types of programs. You specify the type in the Program Properties dialog box for your program definition. The types are:

- Standard, which is a normal entry-point program.
- Upgrade Only, which is used in PeopleSoft upgrade utilities.
- Import Only, which is used by PeopleSoft import utilities.
- Daemon Only, a type of program used as a daemon process.
- Transform Only, a program type used to support Extensible Stylesheet Language Transformations (XSLT).

Daemon Program Type

Application Engine provides a daemon process, called PSDAEMON, that runs continuously when PeopleSoft Process Scheduler is running, and is intended for recurring jobs. It polls the system, checking for certain conditions to occur. A predefined set of conditions is an *event*. When the conditions are true, PSDAEMON schedules a process to handle the event.

PSDAEMON supports limited tracing, because it runs indefinitely. Specifically, it only allows Application Engine tracing at the step and SQL levels, in addition to the standard PeopleSoft SQL and PeopleCode tracing. Other options, such as Timings and DB Optimizer tracing, are not supported.

You activate PSDAEMON in PeopleSoft Process Scheduler or from the command line.

Note. One PSDAEMON process can run for each row in the PS_SERVERDEFN table. The PS_SERVERDEFN.DAEMONENABLED field must be set to 1.

Starting PSDAEMON from the Command Line

The command line syntax is:

```
psdaemon [-CT database_type] [-CD database_name] =>
          [-CO userID] [-CP password] -R server_name
```

Use the **-R** option to query PS_SERVERDEFN, obtaining the daemon group, sleep time, and recycle count (terminate after *N* iterations). *Server_name* is the key value for PS_SERVERDEFN. You do not need to pass ProcessInstance (**-I**) or AE Program ID (**-AI**).

Starting a Daemon Program from PeopleSoft Process Scheduler

Before starting a daemon Application Engine program, you must add the program to the Daemon Group page in PeopleSoft Process Scheduler.

To add a daemon program:

1. Select PeopleTools, Process Scheduler, Daemon Group.
2. Select the Add New Value page.
3. Enter a daemon procedure group name, and click Add.
4. On the Daemon Group page, add the appropriate programs to the program name list.

Restarting the AEDAEMONMGR Program

AEDAEMONMGR is a restartable Application Engine program, which commits after each daemon procedure. When the PSDAEMON executes, it determines whether it must restart AEDAEMONMGR following an abnormal end to a program.

If a restart is not required, PSDAEMON assigns a new process instance and runs AEDAEMONMGR from the beginning. Because of this design, PeopleSoft Process Scheduler does not have to determine whether PSDAEMON exited due to an error or because it had reached the recycle count.

AEDAEMONMGR uses the Daemon Group page value to get related daemon procedures from PS_DAEMONGROUP in order, and then it initiates each procedure. After all procedures have been executed, AEDAEMONMGR logs a sleep message and returns control to PSDAEMON. The sleep time is used only to log an informational message at the end of each cycle, for example "Sleeping for N minutes...". A message is also logged at the beginning of each cycle, so an administrator can monitor the runtime and sleep-time of a specific PSDAEMON process.

If an error occurs in AEDAEMONMGR; if the recycle count has been reached; or if PSSERVERSTAT.DAEMONACTION = '1' (indicating that PeopleSoft Process Scheduler is idle), PSDAEMON exits. Otherwise, it sleeps for the requested number of minutes, and then calls AEDAEMONMGR again.

Using PSDAEMON to Start Parallel Processing

Within a daemon group, programs are invoked sequentially, and one program does not execute until the previous program has completed. The programs contained in a daemon group should be quick programs that scan information to find events. When an event is discovered, the daemon program can use the ProcessRequest class to invoke programs that are not of the daemon type. These *non-daemon type* Application Engine programs can execute in parallel. For that reason, do not include application-specific processing in a PSDAEMON type program.

See Also

Enterprise PeopleTools 8.51 PeopleBook: PeopleSoft Process Scheduler, "Defining PeopleSoft Process Scheduler Support Information," Defining Process Type Definitions

Transform Program Type

Transform Only type programs enable different systems to communicate with one another by transforming messages into appropriate formats. When you specify an Application Engine program as a Transform Only program, you must specify actions of type XSLT or PeopleCode. You can use transform programs to do any of the following:

- Apply a transformation to a message to make its structure comply with the requirements of the target system.
- Perform a data translation on a message so its data is represented according to the conventions of the target system.
- Determine whether to pass a message through to its target, by filtering it based on its content.

See Also

PeopleTools 8.51 PeopleBook: PeopleSoft Integration Broker, "Applying Filtering, Transformation and Translation," Developing Transform Programs Using PeopleSoft Application Engine

Chapter 3

Creating Application Engine Programs

An Application Engine program includes a logically ordered set of sections, steps, and actions. An executable program must contain at least one section, called MAIN, used to identify the starting point of the program; it should contain at least one step; each step should contain at least one action.

This chapter discusses how to:

- View Application Engine programs.
- Filter view contents.
- Print program and flow definitions.
- Create, open, and rename programs.
- Copy or move program elements.
- Test Application Engine programs.
- Set program properties.
- Add sections.
- Add steps.
- Specify actions.

Viewing Application Engine Programs

This section discusses how to:

- Use Definition view.
- Use Program Flow view.
- Switch between Definition and Program Flow views.
- Use the Refresh option.

Using Definition View

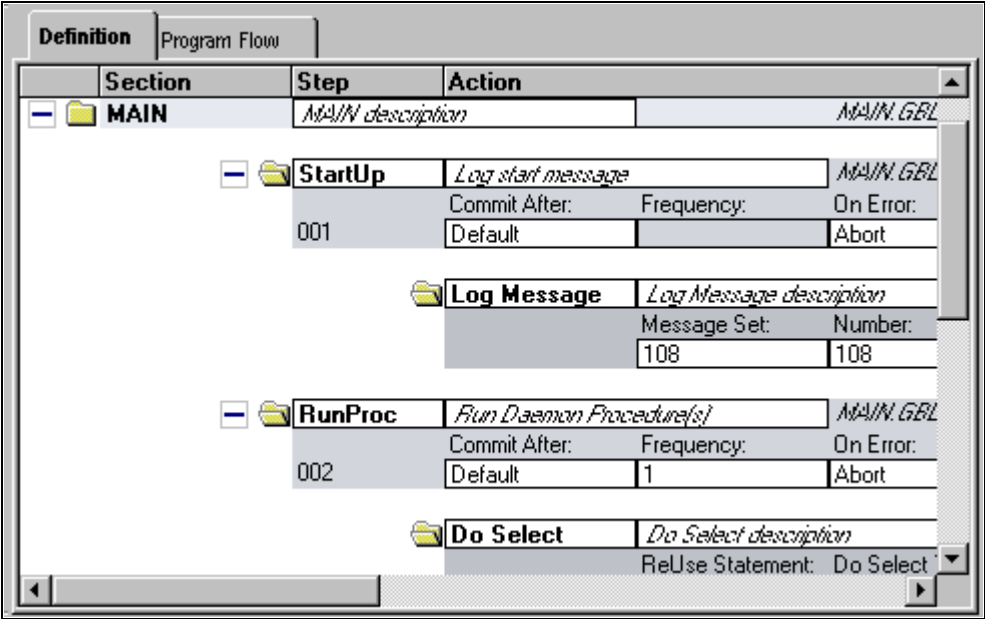
You use Definition view to create definitions within a defined hierarchical structure, in which nodes represent the definitions. A node is the visual representation of a section, step, or action that you can select, collapse, modify, and so on.

The sections that appear in Definition view do not necessarily appear in the order that they execute. To see the actual order in which the sections execute, switch to Program Flow view.

Besides using the mouse, you can navigate in this view using the following keyboard combinations:

- Press Ctrl+Home to scroll to the top of the program definition and select the first node.
- Press Ctrl+End to scroll to the end of the program definition and select the last visible node.
- Press Tab to move from the currently selected field to the next updateable field.
- Press Ctrl+Down Arrow to move from the currently selected node to the next node.
- Press Ctrl+Up Arrow to move from the currently selected node to the previous node.

The following example shows the Definition view:



Example of Application Designer Definition view

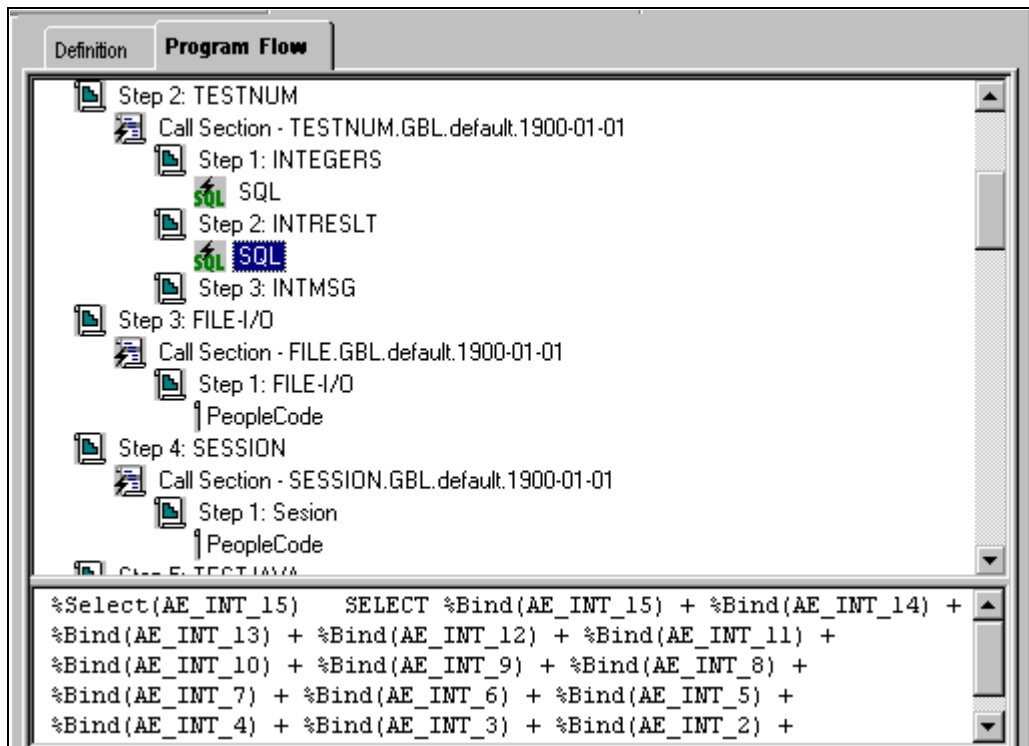
Definition View Pop-up Menu

The following table describes each item you see when you right-click a Definition view window. Certain menu items are enabled only when a particular definition is selected.

Menu Command	Description
View PeopleCode	Launches the PeopleCode Editor with the appropriate PeopleCode loaded. Enabled when a PeopleCode action is selected.
View SQL	Launches the SQL Editor with the appropriate SQL loaded. Enabled when an action containing SQL is selected.
View XSLT	Launches the SQL Editor with the related Extensible Stylesheet Language Transformations (XSLT) text loaded. Enabled for Transform Only program types only, when an XSLT action is selected.
Cut	Removes the selected item and copies it to a clipboard. Here, the word <i>clipboard</i> refers to a PeopleTools-only repository for sharing PeopleTools objects. You cannot copy or paste into another program.
Copy	Copies the selected item.
Paste	Pastes the contents of the PeopleTools clipboard (the most recently cut or copied item) to the current location of the cursor.
Delete	Removes the currently selected node from the program definition.
Refresh View	Refreshes the current view and reorders the definition objects as necessary.
Show Comment	Reveals the comments associated with the selected definition object.
Insert Section	Inserts a new section into the current program, at the place where the cursor is positioned. This option is enabled only when you have MAIN or another section selected.
Insert Step/Action	Inserts a new step and action within the currently selected section. This option is enabled only when you have a section or a step selected.
Insert Action	Inserts a new action within the currently selected step. This option is enabled only when you have a step or action selected.
Jump to This Program Flow	Switches to the Program Flow view with the first occurrence of the currently selected definition in focus.
Print	Displays the print dialog box for the definition view.
Insert Section Into Project	Applies to sections. Inserts the currently selected section into the current project.

Using Program Flow View

Program Flow view is a read-only view that shows the expected sequence of steps to be executed at runtime for the program you are developing. The following example shows the Program Flow view:



Example of Application Designer Program Flow view

You can control the amount of detail that appears for each definition by clicking it to expand to the next level. You can also view the SQL or PeopleCode in the lower (splitter) window area by clicking the lower window.

If a primary step node (one that is not the result of a section call) is selected, the Print Options dialog box permits either the printing to begin at that step node or the printing of the entire program. However, if a secondary step node (one that is the result of a section call), a secondary action node, a call section action node, an SQL node, or a PeopleCode node is selected in the Program Flow view, the Print Options dialog box only permits the printing of the entire program.

To display the pop-up menu for a node, right-click the node. You do not have to select the node first.

You can also display the comments associated with definitions by selecting View, Show All Comments, or for a particular node, right-click and select Show Comment.

You can double-click SQL or PeopleCode statements to launch the editors.

Program Flow Pop-up Menu

The following table describes each pop-up menu item in Program Flow view:

Menu Command	Description
View PeopleCode	Launches the PeopleCode Editor with the appropriate PeopleCode loaded. Enabled when a PeopleCode action is selected.
View SQL	Launches the SQL Editor with the appropriate SQL loaded. Enabled when an action containing SQL is selected.

Menu Command	Description
Refresh View	Refreshes the current view and reorders the definition objects as necessary.
Show Comment	Reveals the comments for a single definition object that appears in the Program Flow view.
Jump to This Definition	Switches to the Definition view with the first occurrence of the currently selected definition object in focus.
Print	Launches the print dialog box for the program view.

Switching Between Definition and Program Flow Views

By default, navigation within either view does not affect the currently active row in the other view. This functionality enables you to retain your place in one view while scrolling around in the other.

To switch between the two views, you can use any of the following methods:

- View tabs.

As with any tabbed interface, if you select a tab, the associated view interface becomes active. When you return to the previous view, it remains positioned on the current or last selected node within the program when you switched. This is true whether you selected the item or just placed the cursor within an edit box.

- View menu.

Select a section or step in the current view (note that selecting an action does not enable this functionality—you can only jump from parent nodes). Then select View, Jump to Program Flow or View, Jump to Definition, depending on the view that is currently active. When you select one of these commands, the focus of the target view depends on what you have selected in the previous view. For example, if you have section C, step 4 selected in Definition view, and you select View, Jump to Program Flow, section C, step 4 is the focus of the Program Flow window. If the selected item is in a program that is not already open, Application Engine opens the appropriate program, and then navigates to the requested node in the view window.

- Pop-up menu.

The same commands as the View menu are also available from the pop-up menu.

Switching Within Program Flow View

While you are in Program Flow view, you can select these options from the pop-up menu:

- Go to Next Reference

Select to switch to the next reference of a particular definition object. This helps you to quickly navigate through a program. For instance, if references to section C, step 4 appear three times because there are multiple calls to this object at runtime, you select Go to Next Reference to quickly and easily navigate to each reference, or call.

- [Jump to this Definition](#)

Select to go directly to the definition node in Definition view that pertains to the current selection in the Program Flow view.

Using the Refresh Option

As you develop an Application Engine program, you may be inserting, renaming, and deleting definitions. In a large program, it can be easy to lose your place or become disoriented. The Refresh option reorders all the nodes for the current definition according to the following logic:

- For standard program definitions, the MAIN section is always displayed first (Library program types do not contain a MAIN section because they contain only callable sections).

The remaining sections appear alphabetically by name, which makes it easier to locate a section within the program definition. The system, at runtime, executes sections through Call Section actions within steps, not by the order in which sections are defined.

- Steps are never automatically reordered in Definition view, and, at runtime, they execute in the sequence in which you define them.
- Actions are always logically reordered within a step, based on their action type, which defines their runtime sequence.

Note. When you save a modified definition, the system automatically refreshes the view.

Application Engine inserts any delete requests for a given section into the current project, regardless of the Tools, Options setting in Application Designer.

For example, suppose you delete a section node from the current Application Engine program, and then you reinsert a section node and rename it to the same name as the section you just deleted. The section object is not inserted into the project regardless of your Tools, Options setting. This is because a delete action already exists for this object. To resolve this situation, either manually remove the delete request before inserting the new copy request or manually reset the proper flags in the upgrade project that changes the action type from delete to copy.

Filtering View Contents

Section filtering options enable you to filter the current view so that you see only sections and steps based on specified criteria.

To enable or modify the filtering options, select View, Section Filtering. You can select from the following filtering options:

<i>Menu Command</i>	<i>Description</i>
No Filtering	Select to see all objects in your program regardless of any section attributes, such as Market, Database Type, Effective Date, and Effective Status.

Menu Command	Description
Default	Select to display the definition filter according to the default filtering criteria. If you change the value of any filter option and click OK, you have defined a custom filter.
Custom	Select to display the definition filter dialog box and define custom filtering options for the current view.

Behavior of Section Filtering Options

When using the section filter options, consider the following:

- The default is no filtering; therefore, all section definitions are included in this view.
- If you select custom filtering, the default filtering options are displayed while you're in the current session of Application Designer.

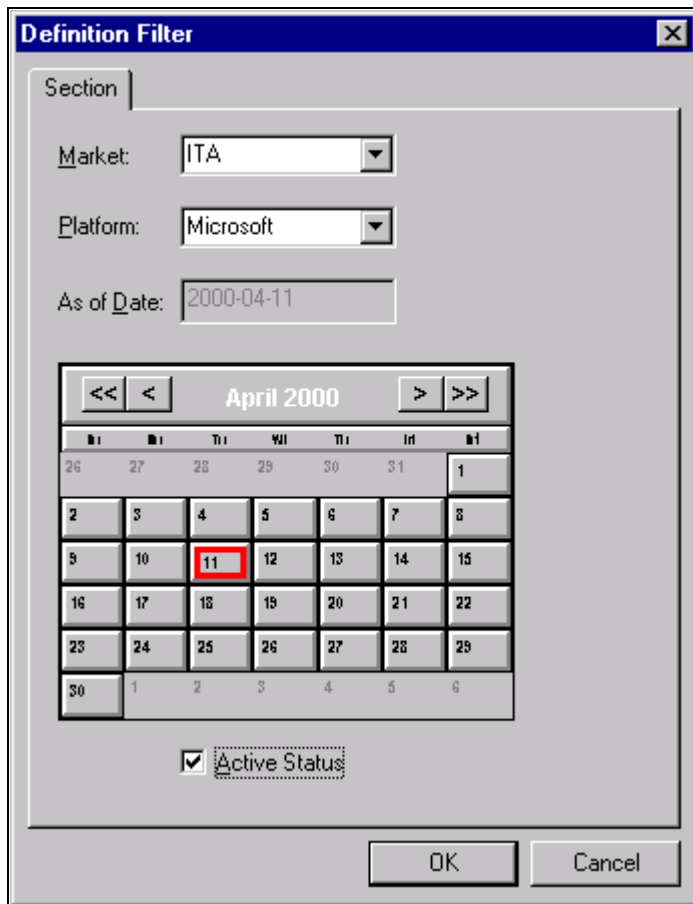
If you modify these filtering options and click OK, the new options are stored as the currently active options and the view is updated accordingly.

- If you select the default filter option, the original default options appear in the dialog box.

After clicking OK, the view reappears with only those sections that qualify. However, if you change the default options and do not click OK, these options are stored as a custom filtering request and the view reappears as necessary.

- If no platform-specific section is defined for the target filter value, the default (base platform) is always included, because this more accurately represents the Application Engine runtime behavior.

If you select Section Filtering, Default, or Section Filtering, Custom, the following dialog box appears:



Definition Filter dialog box

In this example, only definitions that represent the following criteria appear in Definition and Program Flow views.

Market

Select a market code to see only the definitions within that market. To see all market-related definitions for a program, you could update the default profile, or define a custom filter, selecting *(none)* from the Market drop-down list box. In the illustration, sections pertaining only to the Italian market (market code ITA) are shown.

Platform

Select the platform filtering. In the illustration, sections that are defined only for the Microsoft SQL Server platform are shown. Select *Default* to display sections defined to be database-platform-independent (default platform). Specific platforms include Oracle, DB2 UDB for OS/390 and z/OS, DB2/UNIX, Informix, Microsoft, and Sybase.

As of Date

Select the date filtering. In the illustration, sections with an as of date equal to or greater than April 7, 2000 are shown. Select *None* to display all sections, regardless of effective date.

Active Status

Select to show active section definitions.

Note. All filtering options pertain only to section-level nodes.

Printing Program and Flow Definitions

You can print the program definition or program flow depending on which view you are in when you select print.

To print an Application Engine program definition:

1. Right-click and select Print in either Program Flow or Definition view, or select File, Print.
2. Select print options.

Program ID	Select to print the whole program.
All Sections	All sections are expanded in printed report only for the primary Application Engine program being printed and only if that program is an application library. Otherwise, only the MAIN section, first section, or called section is printed.
All Steps	Select to print all the steps in the section.
All Attributes	Select to print all detail level attributes for the specified node and its children.
SQL Statements	Select to print, for every SQL type action, the text of each SQL statement.
PeopleCode Statements	Select to print the text of the PeopleCode statements for every PeopleCode action.
Comments	Select to print the long description comments for the selected node and its children.
Include External Calls	Select to print the section detail of all external calls.
Max No. of Levels (maximum number of levels)	Specify the maximum number of recursive levels to print for the specified call sections, including both external section calls and internal section calls. This edit box is always enabled. The maximum number of levels can only be set to a value greater than or equal to 1.

Creating, Opening, and Renaming Programs

This section discusses how to:

- Create new programs.
- Open existing programs.

- Rename programs.

Creating New Programs

To create a new program definition:

1. Select File, New or press Ctrl + N.
2. In the New dialog box, select *App Engine Program* from the Definition Type drop-down list box, and click OK.
3. Save and name your program.

Select File, Save As, enter the name of your program in the Save Name As edit box, and click OK.

Note. It is also important to provide a program description and specify its owner in the properties dialog box for the new program.

See [Chapter 3, "Creating Application Engine Programs," Setting Program Properties, page 23.](#)

Opening Existing Programs

To open an existing program:

1. Select File, Open.
2. In the Open Definition dialog box, select *App Engine Program* from the Definition Type drop-down list box.
3. Enter your search criteria for the program you want, select your program in the search results list box, and click Open to open the program.

Renaming Programs

To rename a program:

1. Select File, Rename.
2. In the Rename dialog box, make sure that *App Engine Program* appears as the definition type.
3. In the box that contains the results of your search, click the program that you want to rename.
4. Click Rename.
5. Place the cursor in the box that appears around the highlighted program name.
6. Enter the new name for the program.

7. Click Rename again, and respond appropriately in the Confirm Rename dialog box.

Note. The system automatically modifies all static references in other programs to the renamed program. For instance, if you call the renamed program from another Application Engine program, the Call Section action in the calling program is modified to reflect the new program name. All sections and steps are saved under the new name. Only one occurrence of a program name can exist for a given database.

Note. If the renamed program is called in a dynamic Do action, the reference is not automatically modified. You should also manually check and modify any embedded references to the new program name in CallAppEngine or other PeopleCode functions.

Copying or Moving Program Elements

The following procedures apply to sections, steps, and actions. Note that when these functions are performed for a given object, the result applies not only to the selected object, but also includes its defined children, if they exist. Also note that all references to menu items apply not only to the main menu bar items, but also to their related items in the context menu, where applicable.

To copy a definition:

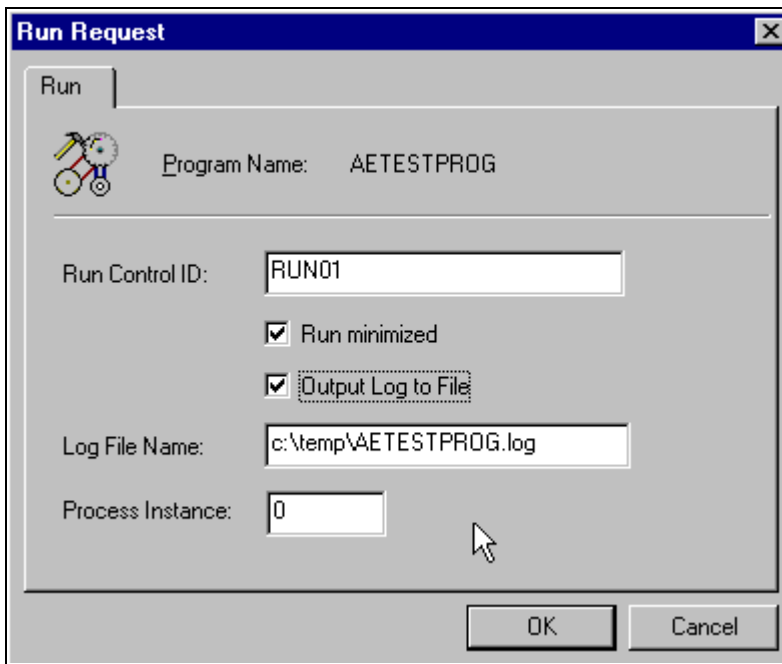
1. Select the definition.
2. Select Edit, Copy.
3. Position the cursor where you would like to put the copied definition, and select Edit, Paste.

To move a definition:

1. Select the definition object.
2. Select Edit, Cut.
3. Position the cursor at the target location and select Edit, Paste.

Testing Application Engine Programs

After creating or modifying your program, you can test it while in Application Designer in two-tier mode. You use the Run Request dialog box:



Run Request dialog box

To run an Application Engine program in two-tier mode:

1. Select Edit, Run Program from the Application Designer toolbar.

The Run Request dialog box appears.

2. Enter appropriate values.

When you click OK, these values are passed as runtime parameters to the initiated Application Engine runtime executable.

Run Control ID	Enter the run control ID of the program that you are testing.
Run Minimized	Select to have the window of the requested process minimized when it is submitted to run.
Output Log to File	Select to write the output log to a file.
Log File Name	Specify the log file name (enabled only when Output Log to File is selected).
Process Instance	Specify the process instance for this run request, or use the default value of zero if an instance number is not needed.

3. Click OK.

Setting Program Properties

This section discusses how to:

- Access properties.
- Set general properties.
- Set state record properties.
- Specify temporary tables.
- Set advanced properties.

Accessing Properties

When you have an Application Engine program open in Application Designer, you can view and modify the properties assigned to an entire program just as you would a step or a section.

To view or modify the properties associated with a program, click the Properties button or select File, Definition Properties while the program is open. You can also press Alt+Enter. The Program Properties dialog box appears.

Setting General Properties

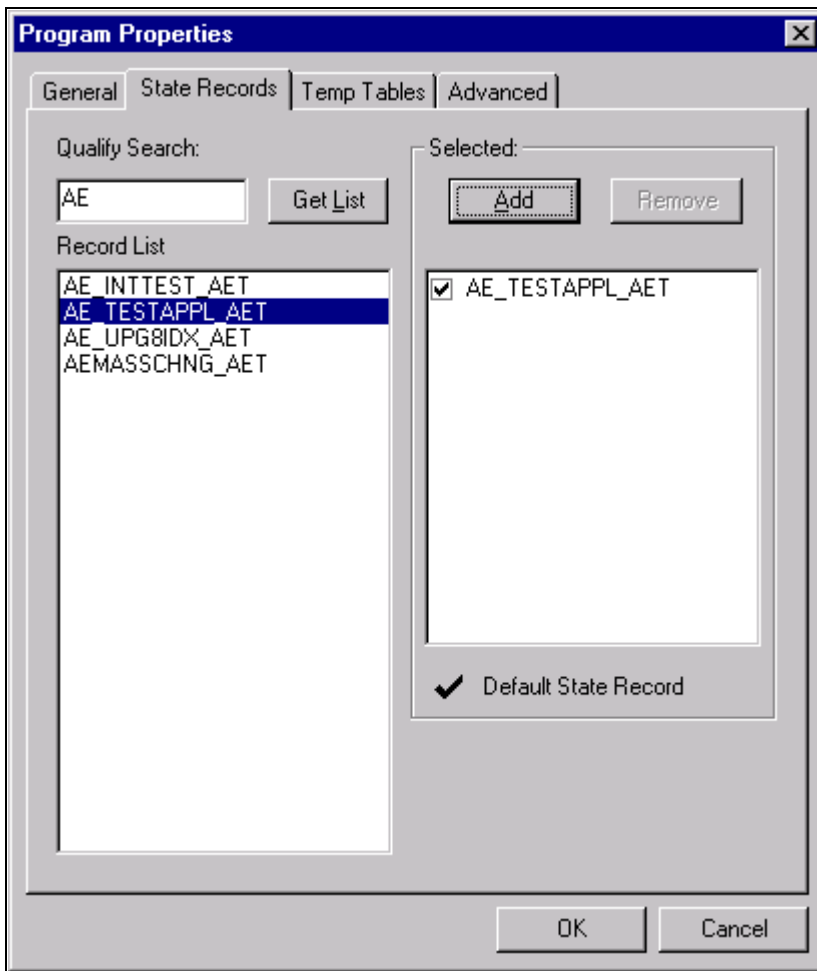
Access the Program Properties dialog box and select the General tab. You can specify identification values for your Application Engine program.

Owner ID

(Optional) Enter the owner ID for the program. The owner ID is a way to identify which definitions are owned by which PeopleSoft applications, such as PeopleSoft General Ledger, Accounts Receivables, and so on. The values in the drop-down list box are Translate table values associated with the OBJECTOWNERID field.

Setting State Record Properties

Select the State Records tab.



Program Properties dialog box: State Records tab

Qualify Search

Enter any wildcard characters or complete table names to limit the results that appear in the record list. By default, the Record List box contains all record names that end with the extension AET. This extension identifies the record as an Application Engine record.

Get List

Click to populate the Record List box.

Record List

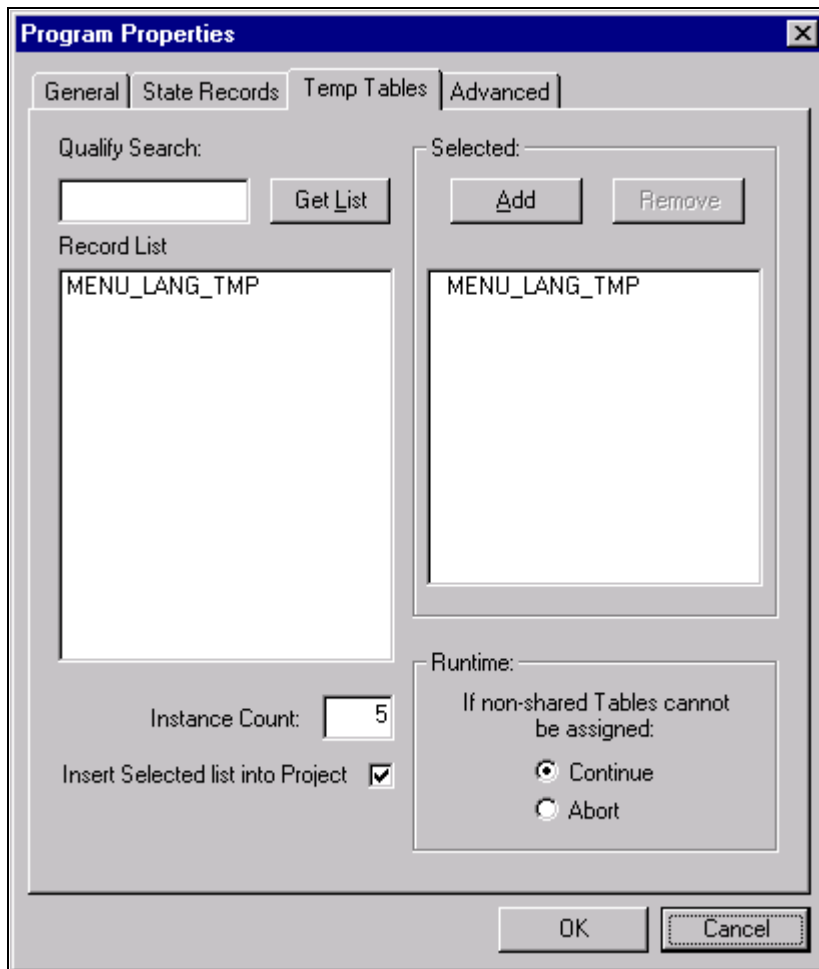
This text box contains the results of your state record search.

Selected

Select state records for use with a particular program. Click Add to include selected records from the record list into the selected list. Click Remove to remove selected records from the selected list. Indicate which state record to act as the default state record by selecting its check box. For your default state record, you need to reference only fieldnames in your PeopleCode and SQL (for the active program). When you reference a non-default state record, you do so by using `recname.fieldname`.

Specifying Temporary Tables

Select the Temp Tables tab.



Program Properties dialog box: Temp Tables tab

Temporary tables store intermediate results during a program run.

Note. You must have already defined required temporary tables in your database prior to associating them with an Application Engine program.

Qualify Search

Enter any wildcard characters or complete table names to limit the results that appear in the record list. By default, the Record List box contains only records that are of type Temporary Table. You apply this attribute when you create the record in Application Designer.

Get List

Click to populate the Record List box.

Record List

This text box contains the results of your search for temporary tables.

Selected	Select temporary tables for use with a particular program. Click Add to include selected records that appear in the record list. Click Remove to exclude selected records that appear in the selected list.
Instance Count	Enter the number of physical tables to be created for each dedicated table for this program during the SQL Build procedure in Application Designer. Typically, you would set this number to equal the maximum number of parallel program runs that you anticipate. For instance, if you expect up to five instances of the same program to run simultaneously, then you would set the instance count to 5.
Insert Selected List into Project	If the active Application Engine program definition belongs to a project, select to include the dedicated temporary tables for this program within the same project.
Runtime	Control how an Application Engine program behaves if an instance of its specified dedicated temporary tables is not available. If you select Continue, then Application Engine uses the base version, or nondedicated version, of the temporary tables. If you select Abort, then the program exits with an error message.

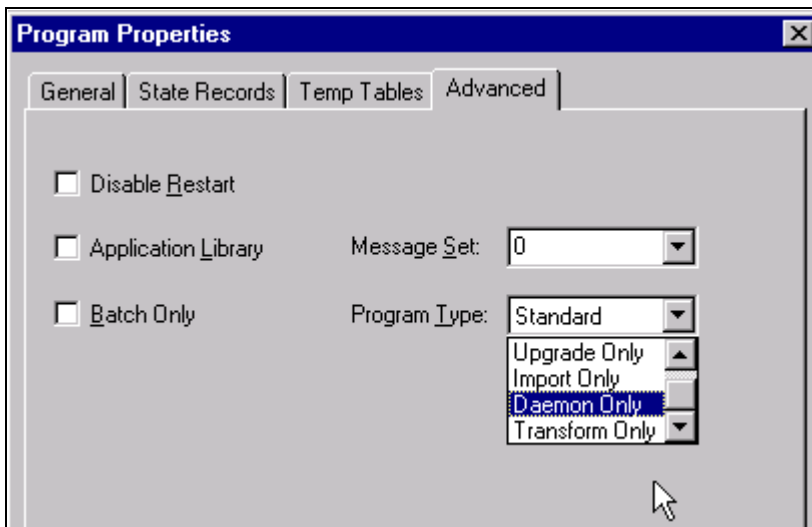
Note. If the table is keyed by PROCESS_INSTANCE, and the application SQL includes the process instance in the Where clause, then the table can be shared by multiple processes. The best performance, however, occurs when a program runs against a dedicated temporary table instance.

See Also

Chapter 9, "Using Temporary Tables," page 161

Setting Advanced Properties

Select the Advanced tab.



Program Properties dialog box: Advanced tab

Disable Restart

Select to disable the built-in restart capabilities for a particular program.

Application Library

In some cases, you may want a program to contain only a collection, or library, of common routines (in the form of callable sections) that you do not want to run as a standalone program. When sections are defined as public, other programs can call the sections, or routines, that exist in the library at runtime. Because this type of program is not designed to run as a standalone program, it does not require the MAIN section, or initial entry point. Select this check box to rename or remove any existing MAIN section.

Note. An application library is the appropriate location to store a collection of shared Application Engine program sections. Libraries are not intended for storing a specific SQL action within a section. To share common SQL, use the SQL repository.

Batch Only

Select for batch-only programs. Batch-only programs are not executed from the CallAppEngine PeopleCode function. Any dedicated temporary table used for batch-only programs do not have online instances created.

Message Set

Specify the default message set value for this program. The system uses this message set value for all Log Message actions where the message set isn't specified.

Program Type

Select from:

- *Standard*: Used by standard entry-point programs.
- *Upgrade Only*: Used by PeopleSoft upgrade utilities only.
- *Import Only*: Used by PeopleSoft import utilities only
- *Daemon Only*: Use for daemon-type programs.
- *Transform Only*: Support for XSLT programs.

Adding Sections

This section provides an overview of sections and discusses how to:

- Insert sections.
- Locate sections.
- Set section properties.

Understanding Sections

A section comprises steps and is somewhat equivalent to a COBOL paragraph or a PeopleCode function. You can create sections that are platform-independent or platform-specific, intended for a particular market, and effective-dated.

Whenever you create a new program, you simultaneously create a section called MAIN. The MAIN section identifies the entry point of the program so that it can be called by another program.

Section Execution Order

A section is unique based on the program and section names, and based on its intended database platform and effective date. You can also create unique market-specific sections. When you execute an Application Engine program, it executes sections based on the following order of precedence:

1. If a section for the current market exists, execute it.
Otherwise, execute the default GBL (global) market section.
2. If a section for the current platform, or database exists, execute it.
Otherwise, execute the default database platform section.
3. If multiple effective-dated sections exists, execute the section with the most recent effective date, based on the current (run) date.

For example, suppose you have two versions of a particular section: SECT01 for the Public Sector market and SECT01 for the Global market. If you request to run the public sector version of the program, Application Engine executes the Public Sector version of SECT01. If the program is running on Oracle, Application Engine then looks for an Oracle version of the SECT01 for Public Sector.

Inserting Sections

To insert a section:

1. Select Insert, Section, or right-click and select Insert Section.

The default name for a section that you insert is Section *N*, where *N* is an incrementally changed number that attempts to provide a unique name for each section object. Unless you rename sections, the sections you add are named Section*N*+1, where *N* is the last section you inserted. Consequently, you get names such as Section1, Section2, Section3, and so on.

The designer inserts the new section directly beneath the subordinate objects within the owning section of the highlighted object. For instance, if Section2 were selected, then Section4 would be inserted *between* Section2 and Section3 rather than *after* Section3.

Note. Sections are always reordered alphabetically by name at save time to make it easier to locate a given section. However, order of execution is dependent on internal call section references and is, therefore, independent of the order that sections are inserted and displayed.

2. Enter the remaining section property values.
3. Save the program.

Locating Sections

Various methods are available for locating references to sections within an entire database as well as within a program.

Finding Call Section References

You can generate a list of all the references to a particular section. The list applies only to Application Engine programs defined within a single database.

To locate section references:

1. Open the program containing the shared, or called, section.
2. Select Edit, Find References.

The Find Definition References dialog box appears.

3. On the Call Sections tab, select the appropriate section from the Section name drop-down list box, or enter the name.

By default, the current program name and MAIN section appears in the dialog box.

4. Click OK.

5. In the output window, view the generated list.

The output window lists the programs and sections that call a particular program. This list also shows the total call references made to a particular section. Call sections within the current program appear first in the list.

Double-click an item in the output window list to automatically navigate the definition view to that calling section.

Finding Sections Within the Current Program

Within large and more complicated Application Engine programs, such as those upgraded from a previous release, it is not uncommon to have over a hundred sections. Rather than scrolling through a large program, use the Go To Section feature.

Note. This feature applies only to the current program.

To automatically navigate to a selected section:

1. Select Edit, Go To Section.


The Find Definition References dialog box appears.

2. On the Go To Section tab, select the appropriate section from the Section name drop-down list box, or enter the name of the section.
3. Click OK.

The Definition view scrolls to the first occurrence of the section with the name you selected.

Setting Section Properties

Controls that specify section properties are located in Definition view. For example, for each section included in your program, a node, as shown in the following example, appears from which you specify all of the attributes to associate with a particular section.

 MAIN	<input type="text" value="MAIN description"/>		MAIN.GBL.(base).1900-01-01				
	Market:	Platform:	Effective Date:	Effective Status:	Section Type:	Auto Commit:	Access:
	<input type="text" value="GBL"/>	<input type="text" value="(base)"/>	<input type="text" value="01/01/1900"/>	<input type="text" value="Active"/>	<input type="text" value="Prepare Only"/>	<input type="checkbox"/> After Step	<input type="checkbox"/> Public

Section object

The values you specify at the section level generally apply to all the objects contained within that section.

- Section Name**

Develop a naming convention and be consistent throughout your projects. You are limited to eight characters.
- Market**

Select the market for which the section is intended. If a particular market is irrelevant to your batch program, keep the default market value of Global (GBL).

Platform	Select the target database platform for which this section definition is to execute. Leave the default value for all sections whose defined actions are not specific to any given database platform.
Effective Date	To make a particular section effective-dated, enter the target date.
Effective Status	Specify whether a section is active or enabled at runtime.
Section Type	<p>In the case of an abnormal termination of the program, the value of this system field specifies whether you must restart the section.</p> <p>If a section controls a procedure that, if not run to completion, could corrupt or desynchronize your data, select <i>Critical Updates</i>. Otherwise, use the default value of <i>Preparation Only</i>.</p>
Auto Commit	Select to specify the commit level for the section. You can have no commit or you can have Application Engine commit after the step successfully completes.
Public	Select to enable a section to be called from another program.

Adding Steps

A step represents the smallest unit of work that can be committed in a program. When you create a program, you have a default MAIN section and step, initially named Step01.

This section discusses how to:

- Insert steps.
- Set up step properties.

Inserting Steps

To insert a step:

1. Highlight the section or step that you want to precede the new step.

For example, if you want the new step to be the first step in the section, select the section node. Otherwise, select the existing step that you want the new step to follow.

Note. The name of the section in which you insert the step appears to the right of the step description. In large programs, this step enables you to determine the section in which a step resides, if the section is not in view. Also, note that a sequence number appears on each step (001, 002, 003, and so on) so that you can determine the order of a step within a section. The sequence numbering for steps begins at 001 within each section.

2. Select Insert, Step/Action.

By default, the steps are given a default name of *StepN+1* beginning with *Step01*. Rename the step to better define the type of actions this step contains.

Note. The designer continues to increment the step name until it has a unique step name within a section. If the designer is unable to create a unique name after 50 attempts, a new step is not inserted.

3. Specify a step name and the remaining values.

To rename the step name, position the cursor in the step name edit box and enter a custom name. Only accept the default name for building quick, simple programs and for training purposes.

Setting Up Step Properties

You set up step properties in Definition view.

Step Name	Enter a name (up to eight characters).
Commit	Specify the commit level for the step: <ul style="list-style-type: none"> • <i>Default:</i> Select to inherit whatever commit level you specified for the section in which the step resides. • <i>Later:</i> Select to postpone the commit until a subsequent commit occurs. Here you can override the section-level commit, if it happened to be set to <i>After Step</i>. • <i>After Step:</i> Select if you have a commit level of <i>None</i> specified at the section level. This way you can override the section-level commit and commit a specific step within a section with no other commits.
Frequency	Enabled only when a step contains one of the following actions: Do While, Do Select, or Do Until. Enter the numeric frequency with which Application Engine should commit. If non-zero, Application Engine commits every <i>N</i> iterations, and then again after the last iteration.

On Error

Specify how Application Engine should respond to an error at the step level. The On Error routine behaves the same for both SQL and PeopleCode actions. The program only terminates on errors, not warnings. Select from:

- *Abort*: The application terminates with an error message.
- *Ignore*: The program continues but logs an error message.
- *Suppress*: The program continues and presents no error message.
- *SQL*: Usually a program terminates if a SQL Prepare statement or execute fails. If you select *Ignore* or *Suppress*, errors on executing programs are suppressed, but errors on compiles still cause the program to terminate. Thus, if you select to reuse on an Update statement, the program fails on the compile if the SQL is incorrect, but it does not fail on a duplicate key error or similar error when the program executes.
- *PeopleCode*: There is a PeopleCode error in the program if the return code satisfies the statement `If (nRet & PCM_ERROR)`.

Status

Select to activate a step. If the step is currently applicable to your program (and working) you'll probably want to keep it active.

Note. The On Error property does not apply to compile errors (for example, specifying erroneous SQL statements). It checks only for execution-type errors. If your program has a syntax error, the program terminates.

Specifying Actions

This section provides an overview of actions and discusses how to:

- Insert actions.
- Set action properties.
- Specify SQL actions.
- Specify Do actions.
- Specify PeopleCode actions.
- Specify Call Section actions.
- Specify Log Message actions.
- Specify XSLT actions.

Understanding Actions

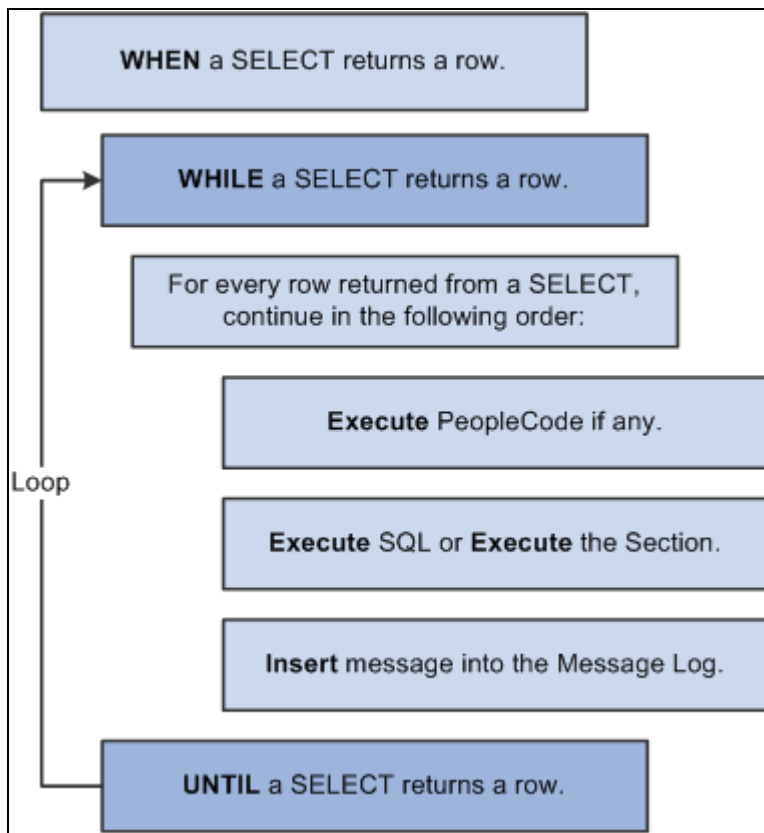
You can include eight types of actions within a step, and a step can contain multiple actions. The actions you define for a step depend on the results that your program requires at each stage of execution.

The only mutually exclusive actions within a single step are Call Section and SQL Statement; you cannot add a Call Section action to a step that already contains a SQL Statement action, and visa versa. You can include only one of each action type within a single step. Because there are eight types of actions, and two are mutually exclusive, the maximum number of actions a single step can contain is seven.

Action Execution Order

At runtime, the system evaluates actions by type and executes them within a strict hierarchy. For example, if both a Do When and PeopleCode action exist within a given step, Application Engine always executes the Do When first.

The following diagram shows the sequence and level of execution for each type of action:



Action execution hierarchy

As you add actions to a step in the definition view, the actions are initially inserted after the selected definition (the owning step or a previous action). However, following a save request or a refresh of the view, the designer reorders all actions to match the execution hierarchy. This feature helps you visualize the sequence in which each step of your program logic executes.

Note. A SQL action and a Call Section action are interchangeable and mutually exclusive. Only one of these two actions can appear within a step.

When inserting actions, remember that:

- You cannot have more than one action of a specific type within the same step.
- You cannot have a SQL action and a Call Section action within the same step.
- You can only define XSLT type actions for programs defined as Transformation types (see the program properties).

Inserting Actions

To insert an action:

1. Highlight the step in which you want to insert the action.
2. Insert the action.

You do this using one of the following methods:

- Select Insert, Step/Action.
 - Right-click the step and select Insert Step/Action.
3. Select the action type from the drop-down list box, or when current action type is selected, type the first character or so of the desired action type, and then press Tab. The first (or only) type qualified by your entry is updated in this control.
 4. Enter a description of the action.
 5. Specify the appropriate properties for the action you selected.

Setting Action Properties

To modify action properties, you must have Definition view active. Because you can include a variety of actions within a step, there are different sets of properties specific to a particular action type. Depending on the action type you select, the properties that appear change.

For example, you can specify the reuse feature with a SQL action. This feature does not apply to a PeopleCode action; instead, you need to specify how to respond to the return value of the PeopleCode program.

This example shows how you can select action-specific properties for different action types:

Do Select	Do Select description
ReUse Statement:	Do Select Type:
No	Select/Fetch

Call Section	Call Section description
Section Name:	Program ID:
STATS	AETESTPROG
<input type="checkbox"/> Dynamic	

Actions and associated properties

PeopleCode and all SQL action types invoke the related PeopleTools Editor to define or maintain the related text.

ReUse Statement Property

The ReUse Statement property is available for all SQL action types (SQL, Do When, Do While, Do Until, Do Select). You use the ReUse Statement property to optimize the SQL in your batch program. ReUse Statement converts any %BIND references to state record fields into real bind variables (:1, :2, and so on), enabling the Application Engine runtime process to compile the statement once, dedicate a cursor, and then re-execute it with new data as often as your program requires. When you are using SQL or a Do action to process a large volume of rows, one at a time, inside a fetch loop, compiling each statement that you issue can be a considerable performance issue. ReUse Statement is a way to combat potential performance decreases.

Note. You can have Application Engine recompile a reused statement by using the %ClearCursor function.

When setting the ReUse Statement option, choose from these values:

- Bulk Insert**

When used in conjunction with statements like `INSERT INTO tablename (field1, field2...) VALUES (%BIND(ref1), %BIND(ref2))`, the Bulk Insert feature offers the most powerful degree of performance enhancements related to the ReUse Statement feature. This option turns on ReUse Statement, and, in addition, it holds all the data in a buffer and performs an insert only after a large volume of rows has gathered in the buffer. The number of rows allowed to gather in the buffer depends on your database platform. Storing data in the buffers is applicable only if you've selected *Bulk Insert* and the SQL is an Insert statement. For statements other than Insert, the Bulk Insert option is ignored.
- No**

Select this option to disable ReUse Statement. With ReUse off, the Application Engine runtime process recompiles the SQL statement every time the loop executes. By default, ReUse Statement is disabled.
- Yes**

Select this option to enable basic ReUse Statement functionality.

Note. The ReUse Statement property can offer significant performance gains. However, do not use it if %BIND variables are building parts of the SQL statement or are in the field list of a Select statement (this does not apply if you use the Static option in %BIND).

Specifying SQL Actions

This is the default action type for the first action within a given step. Use this action to perform the following SQL commands on multiple rows:

- Update
- Insert
- Delete
- Select

Note. Before you insert SQL (select View, SQL) into a SQL action within a new Application Engine program, you must have previously saved the program. This is required because the program name you use to save this definition is used to relate your program with the SQL objects you are about to create. The same is true for inserting PeopleCode.

With a SQL action, you use the SQL Editor to create and modify a SQL statement. Following are some examples of SQL statements:

```
%Select(AF_PERFM_AET.PREV_ASOF_DT)
SELECT %DateOut(ASOF_DT)
FROM PS_AF_FCST_SCHT%Bind(EPM_CORE_AET.TABLE_APPEND,NOQUOTES)
WHERE AFDEFN_ID = %Bind(AF_CORE_AET.AFDEFN_ID)
AND ASOF_DT = (SELECT MAX(ASOF_DT)
FROM PS_AF_FCST_SCHT%Bind(EPM_CORE_AET.TABLE_APPEND,NOQUOTES)
WHERE AFDEFN_ID = %Bind(AF_CORE_AET.AFDEFN_ID)
AND ASOF_DT < %Bind(AF_PERFM_AET.ASOF_DT))
```

Note. If you intend to include multiple SQL statements within a single action, you should use the meta-SQL construct %EXECUTE. The previous sample SQL statement sample contains bind variables from a previous Application Engine action.

No Rows Property

In addition to the ReUse Statement property, the No Rows property is available for SQL actions. If the SQL (Insert, Update, or Delete) associated with the SQL action does not return any rows, you must specify what the Application Engine program should do.

For example, you could use this in a case where you insert into a temporary table, and then you intend to perform further operations on the inserted rows (provided that some rows meet the criteria). If the initial combination of Insert and Select statements provides no rows, you could save the program from having to reselect on the temporary table before executing another operation, or you could prevent the program from performing set operations on the table with no qualifying rows.

When you set the No Rows property, you choose from the following values:

Abort The program terminates.

Section Break Application Engine exits the current section immediately, and control returns to the calling step.

<i>Continue</i>	The program continues processing.
<i>Skip Step</i>	Application Engine exits the current step immediately and moves on to the next step. Application Engine ignores the commit for the current step at runtime. If the current step contains only one action, use Skip Step only to bypass the commit.

Note. Using No Rows in conjunction with a Truncate Table operation is unreliable. Some database platforms report zero rows affected for truncations, regardless of how many rows were in the table.

Specifying Do Actions

There are four types of Application Engine actions that, although distinct from the others, can be grouped together as:

- Do When
- Do While
- Do Until
- Do Select

Use these actions to control the execution of your program. With these action types, you can control the execution of subsequent sections, actions, or SQL statements depending on the results of a Do SQL statement in the form of a Select statement. If you were coding in COBOL, you would perform similar actions using the If and While functions.

Any of the Do actions can control the execution of a section, a SQL statement, a PeopleCode program, or a log message. For example, a Do Select can execute a SQL statement for each row returned by the included Select statement.

Do When

When using a Do When action, consider the following:

- The Do When action is a Select statement that allows subsequent actions to be executed if any rows of data are returned.
- This action is similar to a COBOL If statement.

A Do When statement runs before any other actions in the step. If the Do When statement returns any rows, the next action is executed. If the Do When conditions are not met, the remaining actions within that step are not executed. Your program executes a Do When action only once when the owning step executes.

- The only property that you can specify for the Do When action is the ReUse Statement property, which applies to all SQL-based actions.

Do While

The Do While action is a Select statement that, if present, runs before subsequent actions of the step. If the Do While does not return any rows of data, the action terminates. The Do While is identical to the COBOL While statement. Subsequent actions within the step are executed in a loop as long as at least one row is returned by the Select statement for the Do While action. If the Do While does not return any rows, the step is complete.

The only property that you can specify for the Do While action is the ReUse Statement property, which applies to all SQL-based actions.

Do Until

A Do Until action is a Select statement that runs after each action when a step completes. If the Select statement returns any rows of data, the step terminates.

- Use a Do Until action if you want the processing actions to execute at least once, and to execute over and over until a certain condition is true, such as until a Select statement returns some rows.
- You can also use a Do Until action to stop a Do Select action prematurely.

For example, if the Select statement for the Do Until action does not return any rows, then the actions in the step are repeated (except if a Do When action appears in the step). Normally, a Do Select action continues until no rows are returned. If any rows of data are returned, the Do Select action stops and the step is not repeated.

- The only property that you can specify for the Do Until action is the ReUse Statement property, which applies to all SQL-based actions.

Do Select

The Do Select action is a Select statement that executes subsequent actions once for every row of data that the Do Select returns. For instance, a Do Select can execute a SQL statement for each row returned from the Select statement. The subsequent actions within the step are executed in a loop based on the results of the Select statement. The type of the Do Select determines the specific looping rules.

Like the other Do actions, for the Do Select action you can specify the ReUse Statement property, which applies to all SQL-based actions.

In addition to the ReUse Statement property, you must also specify another Do Select property: *Do Select Type*.

Note. Application Engine does not commit a step containing a Do Select action with the Select/Fetch option enabled until the entire step completes successfully, regardless of the other options you have selected.

For example, suppose at the step level, you specified to commit every 100 iterations of the step. One of the actions of this step is a Do Select with Select/Fetch chosen. Because Application Engine does not checkpoint or commit while the Do Select is active, the transaction performed by the actions within a step is not committed until the entire step completes successfully. This is also true if any sections are called from inside the loop.

Do Select Type Property

When you specify the Do Select Type property in a Do Select action, you choose from the following values:

Select/Fetch

Application Engine opens a cursor for the Do Select action, and then, within that cursor, Application Engine performs a Fetch statement for each iteration of the loop to get each row from the Select statement. When a Fetch statement results in an end of table message, the looping is complete. You can't restart this type of Select statement, because Application Engine does not perform a checkpoint or a commit within the step containing this action while Select/Fetch is running. Ultimately, your program ignores the commit settings at runtime until the outermost Select/Fetch completes.

Note. When an Application Engine program is not set up for the capability to restart, then commits are not controlled, monitored, or restricted by Application Engine. When Restart is disabled, commits are controlled by the program.

Re-Select

For each iteration of the loop, Application Engine opens a cursor and fetches the first row. Your program processes the first row returned from the Select statement. The cursor is reopened for each iteration of the loop. With this type of Fetch statement, you typically want some aspect of the loop to eventually cause the Select statement to return no rows. Otherwise, there is no mechanism in place by which to exit the loop. This type of Do Select is restartable.

Restartable

This option is similar to *Select/Fetch* in that Application Engine opens the cursor associated with the Do Select action once, and then it performs a Fetch statement on each iteration of the loop to get each row from the Select statement. However, unlike the *Select/Fetch* option, you can restart this action, because Application Engine performs a checkpoint in the middle of the step. Application Engine treats this loop as if it is restartable, but it does not manage the restart. Make sure that the SQL you include within this action is such that, upon restart, the program recognizes where the previous run failed and where to restart processing. For example, you can employ a processed switch, or base the next Select statement on the key.

Specifying PeopleCode Actions

Use this action type to insert PeopleCode within your Application Engine program. You can invoke the PeopleCode Editor directly from the designer interface to code your PeopleCode programs.

With a PeopleCode action, there is only one property that you can specify—On Return.

Use the On Return value to determine how your Application Engine program reacts based on the return of your PeopleCode program. The On Return setting takes effect if your PeopleCode program issues a "return 1" or "exit 1." You can use the True keyword in place of a non-zero numeric return.

When you specify the On Return property, you choose from the following values:

<i>Abort</i>	The program issues an error and exits immediately.
<i>Break</i>	The program exits the current step and section, and control returns to the calling step.
<i>Skip Step</i>	The program exits the current step, and continues processing at the next step in the section. If this is the last step in the section, the calling step resumes control of the processing.

Specifying Call Section Actions

Use the Call Section action to call another section defined in an Application Engine program. You can call a local section defined within your current program, and you can make external calls to a section defined in another Application Engine program.

The external section you intend to call must have its access property set to Public. If a section's access property is set to Private, that section can be called only from within the same program. By default, a section's access property is Private. If you attempt to make a call to a section that does not allow external calls, you receive an error message at runtime.

Note. You can call only programs that reside within the same database as the calling program.

Program ID Property

Because you can call sections defined in the current program or within external programs, you must first specify the program ID of the program containing the section you intend to call.

The default value is (*current*). If you call a section defined in another program, make sure that you first select the appropriate external program from the Program ID drop-down list box. The drop-down list box contains the names of all program definitions that currently exist in the database.

Section Name Property

Select from names defined in the program that appears in the Program ID list box. To call a section that is defined in an external program, select the program name in the Program ID edit box prior to selecting the section name.

Also use the Call Section action to call an entire external program. First, select the program ID, and then select section name MAIN. At runtime, this call executes the entire program defined by the value in the Program ID field.

Note. Application Designer does not prevent you from calling the main section of the current program or the current section. For instance, Section1 can contain a step that has a local call section reference for Section1. This enables recursive calls, and should therefore be used with caution.

Dynamic Property

Use the AE_APPLID and AE_SECTION fields in the state record to execute different sections depending on the conditions a program encounters during runtime.

These two fields must be defined on the default state record for the program. If AE_APPLID is not present or is blank (at runtime), the current program is substituted for the AE_APPLID value. If AE_SECTION is not present or is blank, an error occurs.

When issuing a dynamic call, both the section and the program ID must be dynamically set. You enable a dynamic call by first having your program store different section names in the AE_SECTION field, and different program names in AE_APPLID field. The values you insert in these fields are normally based on various conditions met within your program. You then create a Call Section action that calls the section name defined in the state record field by selecting the Dynamic check box.

Selecting Dynamic automatically populates the AE_SECTION field with the symbolic value %Section, and the Program ID field with the symbolic value %AEAPPLID. At runtime, the program calls the section name stored in AE_SECTION that belongs to the program name defined by AE_APPLID.

Program Properties of Called Sections

When you call a section defined in an external program, the current program (the program containing the defined call section) defines the properties that apply to the running process. Suppose tracing is enabled for the current program, but tracing is disabled for the called program section. In this case, the called program has the trace option enabled at runtime because it inherits the properties of the calling program.

For example, if program A calls program B, and program B calls program C, then the properties of A apply to both programs B and C. The calling program always controls the properties for the called program. In this case, program A controls the properties for program B, and because program B inherits the properties of program A, when program B calls program C, program A's properties also apply to program C.


Note. Although program properties are inherited, state records do not follow this inheritance model.

State Records of Called Programs

When you call a program from another program, the called program's default state record becomes active until processing returns to the initial program. However, all of the state records associated with both programs are available. State records that are common between the two programs share values. To communicate between the two programs, or share %BIND variables, define the same state records in both programs.

Specifying Log Message Actions

Use this type of action to write a message to the message log. The message log refers to the PeopleTools table (PS_MESSAGE_LOG) where execution messages reside. Any substitution parameters are written to PS_MESSAGE_LOGPARM. The following illustration shows a Log Message action:

	Log Message			<i>Log Message description</i>
	Message Set:	Number:	Parameters:	
	10662	278	%BIND(AF_CORE_AET.AFDEFN_ID)	

Log Message action

You can use the Log Message action to insert any type of messages. Typically, a Log Message action writes error messages to the message log, but you could also write informational or status messages.

Note. You can also use MessageBox PeopleCode to populate PS_MESSAGE_LOG instead of using the Log Message action. This enables you to easily record errors encountered within Application Engine PeopleCode programs.

Message Set and Number

Select the message defined in the message catalog.

Parameters

Enter values to insert in the log message. This field should be a comma-delimited list of values to substitute for the message variables (%1, %2, and so on) in the message text. These parameters can be hard-coded values or %Bind references. The information specified is inserted in the PS_MESSAGE_LOG at runtime, and any %Bind values are replaced by the current state record field values. You can then view the logged messages from the Process Monitor page.

For example, using message set 1012, number 10, the message reads "The total number of %1 rows exceeds the control count value, %2," and you need the following parameters:

Invoice, %Bind(CONTROL_CNT)

Suppose you run this program with the CONTROL_CNT field value of 120. When the process ends, the following message would be included on the Process Details dialog box in Process Monitor: "The total number of Invoice rows exceeds the control count value, 120."

Specifying XSLT Actions

These are used for transform programs only.

See Also

PeopleTools 8.51 PeopleBook: PeopleSoft Integration Broker, "Applying Filtering, Transformation and Translation," Using XSLT for Transformation

Chapter 4

Developing Efficient Programs

This chapter discusses how to:

- Use state records.
- Set commits.
- Reuse statements.
- Use Bulk Insert.
- Use set processing.

Using State Records

This section provides an overview of state records and discusses how to:

- Share state records.
- Choose a record type for state records.

Understanding State Records

You assign variables for your Application Engine program through state records, while sections, steps, and actions pass values to subsequent program steps through state records.

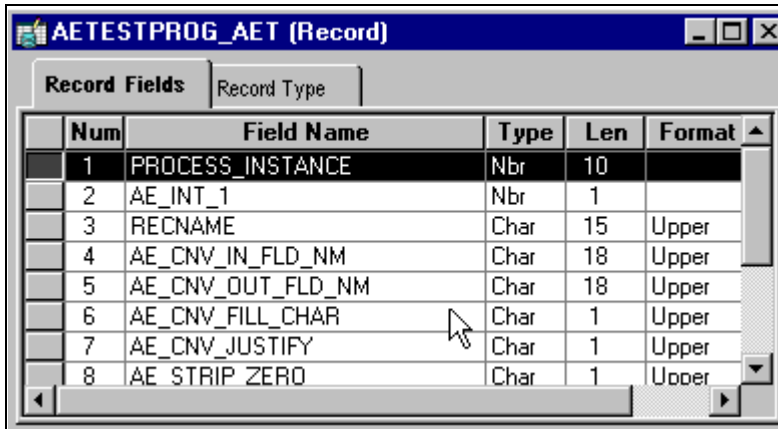
You can have up to 200 state records associated with a particular Application Engine program. However, only one record can be the default state record. You can specify both work (derived) and physical (SQL table) records to be used as state records. The only difference is that derived state records cannot have their values saved to the database at commit time, and so the values are lost during a restart. Therefore, Application Engine erases the contents of derived state records at commit time if Restart is enabled for the current process.

A Application Engine state record must have a process instance defined as the first field and the only key field, and the state record name must end with _AET.

Not all the database columns referenced in your program must be in the state record, just the columns that must be selected into memory so those values can be referenced in a subsequent program action. You may also want to include additional fields to hold pieces of dynamic SQL, to use as temporary flags, and so on.

Application Engine supports long fields, unlike COBOL or Structured Query Reports (SQR). However, it allows only one long field per state record. You set a maximum size for the field in Application Designer and make sure that the data space is compatible with the size of the field that you set.

Application Engine also supports image fields and long text fields.



The screenshot shows a window titled "AETESTPROG_AET (Record)". Inside, there are two tabs: "Record Fields" (selected) and "Record Type". Below the tabs is a table with the following data:

Num	Field Name	Type	Len	Format
1	PROCESS_INSTANCE	Nbr	10	
2	AE_INT_1	Nbr	1	
3	RECNAME	Char	15	Upper
4	AE_CNV_IN_FLD_NM	Char	18	Upper
5	AE_CNV_OUT_FLD_NM	Char	18	Upper
6	AE_CNV_FILL_CHAR	Char	1	Upper
7	AE_CNV_JUSTIFY	Char	1	Upper
8	AE_STRIP_ZERO	Char	1	Upper

Sample state record

During batch processing, Application Engine automatically performs all state record updates. When a program starts, it inserts a row into the state record that corresponds to the process instance assigned to that program run. Application Engine updates the record whenever a commit operation occurs. When restart is enabled and a commit occurs, all state records that have been updated in memory are written to the database, except for derived state records, which are initialized instead.

After the program completes successfully, Application Engine deletes the corresponding row in the state record. There is only one row in the state record for each process instance. Multiple programs can use the same state record, and each program has its own row based on the unique process instance key.

To set values in the state record, you use the %SELECT construct in a SQL statement or write PeopleCode that references the state field with the standard record.field notation. To reference fields in the state record, use the %BIND construct.

Sharing State Records

State records can be used by multiple sections and by multiple programs. When you call a section in another program, any additional state records defined for that program (as in state records that are not already in use by the calling program) are initialized, even if the program has been called previously during the run. However, state records that are common to both programs retain their current values.

To reference variables that exist within a state record, use the following:

```
%BIND(fieldname)
```

Unless a specific record name is specified preceding the fieldname, %BIND references the default state record. To reference a state record other than the default, use the following:

```
%BIND(recordname.fieldname)
```

In the case of a called program or section, if the called program has its own default state record defined, then Application Engine uses that default state record to resolve the %BIND(fieldname). Otherwise, the called program inherits the default state record of the calling program. In theory, the called program does not require a state record if all the fields it needs for processing exist on the calling program's state record.

For those state records that are shared between programs (during an external call section), any changes made by the called program remain when control returns to the calling program. Any subsequent actions in the calling program can access residual values left in the common state records by the called program. This can be useful to return output values or status to the calling program, yet it can also cause unforeseen errors.

Generally, a called program should not share state records with the caller unless you need to pass parameters between them. Most programs have their own set of state records unless a program calls another program that requires specific input or output variables. In that case, you must include the state record of the called program into the calling program's state record list, and make sure to set the input values before issuing the call section.

Choosing a Record Type for State Records

As a general rule, to preserve state record field values across commits in your program, you should store those values in a state record with a record type of SQL Table. Only derived/work-type state records store values that don't need to be accessed across commits. Derived/work records are, however, an excellent choice for temporary flags and dynamic SQL containers that are set and then referenced immediately. Because these values aren't needed later, you don't want to have to save them to the database at each commit. When you create your state record in Application Designer, you should have an idea regarding how your state record will be used. With this information, you can select the appropriate record type to build.

With Application Engine programs, state records that are derived/work records function the same as SQL Table records. However, there is one notable distinction: unless you have disabled Restart, derived work records have their field values reinitialized after each commit. Therefore, unless you anticipate this behavior, you may encounter problems. One quick way to diagnose such a problem is to examine a trace. Typically, you see %BIND variables resolved to values prior to a commit, and then after the commit, they have no value.

This behavior is necessary to ensure consistency in the event of an abnormal termination and restart. During the restart, Application Engine begins, or restarts, at the point of the last successful commit and restores the values of any state records with corresponding database tables. Derived/work records aren't associated with a physical database table, and consequently they can't be restored in the event of a restart.

Setting Commits

For new Application Engine programs that you develop, by default, the commit values at the section and the step level are turned off. No commits occur during the program run, except for the implicit commit that occurs after the successful completion of the program.

You are responsible for dividing your program into logical units of work by setting commit points within your program. Typically, a good time to commit is after Application Engine completes a self-contained task. How often you apply commits affects how your program performs in the event of a restart. For set processing programs, commit early and often. For row-based processing, commit after every *N* iterations of the main fetch loop that drives the process.

If you have a step with a Do While, Do Until, or a Do Select action, you can set the frequency option, which drives your commit level. This setting enables you to set a commit at the step level that occurs after a specified number of iterations of your looping construct. Application Engine programs commit whenever they are instructed to do so, so you can enable the frequency option as well as have other individual commits inside of a loop.

The only restriction for batch runs occurs when you have restart enabled, and you are inside a Do Select action that is of the Select/Fetch type (instead of Re-select or Restartable). With Select/Fetch, all commits inside the loop are ignored, including the commit frequency if it is set.

The Restartable option is similar to Select/Fetch, except that you are implying to Application Engine that your SQL is structured in such a way that it filters out rows that have been processed and committed. This enables a successful restart. One technique for accomplishing this is to have a processed flag that you check in the Where clause of the Do Select action, and you perform an update inside the loop (and before the commit) to set the flag to Y on each row that you fetch.

The commit logic is designed to perform a commit regardless of whether any database changes have occurred. The program commits as instructed, except when the program is restartable and at a point where a commit would affect restart integrity—inside a non-restartable Do Select action, for example.

When you set a step to commit by default, the commit frequency of the step is controlled by the auto commit setting of the section. If the section is set to commit after every step, then the program commits. Otherwise, the program never commits unless the step is explicitly set to commit afterward.

Note. The Commit After, Later setting at the step level enables you to override the section setting if you do not want to commit after a particular step.

%TruncateTable Considerations

Some databases, such as Oracle, issue an implicit commit for a truncate command. If there were other pending (uncommitted) database changes, the results would differ if an abend occurred after the %TruncateTable. To ensure consistency and restart integrity, Application Engine checks the following:

- Whether there are pending changes when resolving a %TruncateTable.
- If the program is at a point where a commit is not allowed.

If either condition is true, Application Engine issues delete from syntax instead.

Considerations with the No Rows Setting

The default for the No Rows setting (on the action) is *Continue*. This setting controls how your program responds when a statement returns no rows. In the case of %UpdateStats, you may want to set No Rows to *Skip Step* and thus skip the commit. For example, suppose you have a single Insert statement into a table, followed by an %UpdateStats. If the stats were current before the Insert statement, and the Insert statement affects no rows, then the %UpdateStats is unnecessary.

Reusing Statements

One of the key performance features of Application Engine is the ability to reuse SQL statements by dedicating a persistent cursor to that statement.

Unless you select the ReUse property for a SQL action, %BIND fields are substituted with literal values in the SQL statement. The database has to recompile the statement every time it is executed.

However, selecting ReUse converts any %BIND fields into real bind variables (:1, :2, and so on), enabling Application Engine to compile the statement once, dedicate a cursor, and re-execute it with new data multiple times. This reduction in compile time can result in dramatic improvements to performance.

In addition, some databases have SQL statement caching. Every time they receive SQL, they compare it against their cache of previously executed statements to see if they have seen it before. If so, they can reuse the old query plan, but only if the SQL text matches exactly. This circumstance is unlikely with literals instead of bind variables.

When using ReUse, keep the following items in mind:

- ReUse is valid only for SQL actions.
- Use ReUse only if you do not use bind variables for column names.
- Use ReUse only if you have no %BIND variables in the Select list.
- If the SQL is dynamic, as in you are using %BIND to resolve to a value other than a standard bind value, and the contents of the bind change each time the statement is executed, then you cannot enable ReUse.

In this situation, the SQL is different each time (at least from the database perspective) and, therefore, cannot be reused.

- If you use the NOQUOTES modifier inside %BIND, a STATIC is implied.

For dynamic SQL substitution, the %BIND has a Char field and NOQUOTES to insert SQL rather than a literal value. If you enable ReUse, the value of the Char field is substituted inline, instead of using a bind marker (as in :1, :2, and so on). The next time that the action executes, the SQL that it executes is the same as before, even if the value of a static bind has changed.

- To prepare a reused statement from scratch, because one of the static binds has changed and the SQL has to reflect that, use %ClearCursor.
- When making calls to an external section, program or library, the reusable cursors are retained upon exiting the program. However, if the calling program attempts to call another external section thereafter, the reusable cursors are discarded.

If you are running DB2 on OS/390 or AS/400, use the ReUse property only when you are not using %BINDS as operands of the same operator, as shown in the following example:

```
UPDATE PS_PO_WRK1
SET TAX = %BIND(STATE) + %BIND(FED)
```

This example causes error -417. You can modify the SQL so that you can use ReUse successfully. Suppose your program contains the following SQL:

```
UPDATE PS_PO_WRK1
SET TAX = 0
WHERE %BIND(TAX_EXEMPT) = %BIND(TAX_STATUS)
```

If you modify it to resemble the following SQL, ReUse works:

```
UPDATE PS_PO_WRK1
SET TAX = 0
WHERE %BIND(TAX_EXEMPT, STATIC) = %BIND(TAX_STATUS)
```

Using Bulk Insert

By buffering rows to be inserted, some databases can get a considerable performance boost. Application Engine offers this nonstandard SQL enhancement on the following databases: Oracle, Microsoft SQLServer, and DB2. This feature is named Bulk Insert. For those database platforms that do not support bulk insert, this flag is ignored.

You should consider using this feature only when an Insert SQL statement is called multiple times in the absence of intervening Commit statements.

Application Engine ignores the Bulk Insert setting in the following situations:

- The SQL is not an Insert statement.
- The SQL is other than an Insert/Values statement that inserts one row at a time.

For instance, the following statements are ignored: Insert/Select, Update, or Delete.

- The SQL does not have a Values clause.
- The SQL does not have a field list before the Values clause.

Note. Bulk Insert is also ignored when all three of the following conditions are true: the database platform is Oracle, the record contains an EFFDT field (effective date), and the record contains a mobile trigger. A mobile trigger is required because an Oracle database does not allow the reading of mutating tables in a row trigger.

In the situations where the Bulk Insert setting is ignored, Application Engine still executes the SQL; it just doesn't take advantage of the performance boost associated with Bulk Insert.

To prepare or flush a Bulk Insert statement because one of the static binds has changed and the SQL has to reflect that, use %ClearCursor. A flush also occurs automatically before each commit.

Using Set Processing

This section provides an overview of set processing and discusses how to:

- Use set processing effectively.
- Avoid row-by-row processing.
- Use set processing examples.

Understanding Set Processing

Set processing is an SQL technique used to process groups, or sets of rows, at one time rather than processing each row individually. Set processing enables you to apply a business rule directly on the data (preferably while it resides in a temporary table) in the database using an Update or Insert/Select statement. Most of the performance gain is because the processing occurs in the database instead of loading the data into the application program, processing it, and then inserting the results back into the database tables. Because the data never leaves the database with set processing (whether it remains in the same table), you effectively eliminate the network round trip and database API overhead.

Note. Because the updates in set processing occur within the database, use temporary tables to hold transient data while your program runs. Although temporary tables are not required for set processing, they are often essential to achieve optimum performance in your batch program.

Using Set Processing Effectively

Keep the following in mind if you are developing new or upgrading older Application Engine programs to adhere to a set-based model.

SQL Expertise

Even if you are developing row-by-row programs with Application Engine, you should be a SQL expert. With set-based programs, this is especially true. The following concepts are particularly important:

- Group by and Having clauses.
- Complex joins.
- Subqueries (correlated and non-correlated).
- Tools for your database to analyze complex SQL statements for performance analysis.

Typically, you use these SQL constructs to refine or filter the set to contain only the rows that meet particular criteria. Keep in mind that SQL is what you code with in Application Engine, and Application Engine passes that SQL directly to the database, where it is processed. If you have a complex SQL statement that works functionally, it may not necessarily perform well if it is not properly tuned.

Planning

Well-constructed, robust, and efficient Application Engine programs are usually the product of a detailed planning stage where loops, program flow, the use of temporary tables, sections, steps, and so on, are discussed.

In an ideal situation, address batch processing as a whole while you are designing the system. Sometimes, systems analysts and developers focus primarily on the online system during the database design, and then they consider the batch component within the existing database design. Set processing works best in an environment where the data models are optimized for set processing.

For example, you could have a separate staging table for new data that hasn't been processed, rather than having numerous cases where existing rows in a table get updated. In set processing, it is much easier to process the data after moving it to a temporary table using an Insert or Select statement rather than just using an update. Avoid performing updates on real application tables, and try to perform your updates on temporary tables. To minimize updating real application tables, structure your data model to prevent that.

Another important consideration is keeping historical data separate from active transactions. After the life cycle of given piece of transaction data is over, so that no more updates are possible, consider moving that data to an archive or history table and deleting it from the real transaction table. This keeps the number of rows in the table to a minimum, which improves performance for queries and updates to your active data.

Temporary Tables

Although temporary tables are not required for set processing, well-designed temporary tables complement your set-based program in a variety of ways.

Creating temporary tables enables you to achieve one of the main objectives involved with set based processing—the processing remains on the database server. By storing transient data in temporary tables, you avoid the situation where the batch program fetches the data, row by row, and runs the business rule, processes the data, and then passes the updated data back to the database. If the program were running on the client, you encounter performance issues due to the network round trip and the diminished processing speed of a client compared to the database platform.

Your temporary tables should be designed to accomplish the following:

- Hold transaction data for the current run or iteration of your program.
- Contain only those rows of data affected by the business rule.
- Present key information in a denormalized, or flattened, form, which provides the most efficient processing.
- Switch the keys for rows coming from the master tables if needed.

A transaction may use a different key than what appears on the master tables.

Denormalized Tables

The most efficient temporary tables store data in denormalized form. Because most programs need to access data that resides in multiple tables, it is more sensible to consolidate all of the affected and related data into one table, the temporary table. It's much more efficient for the program to run directly against the flattened temporary table rather than relying on the system to materialize complex joins and views to retrieve or update necessary data for each transaction.

If your program requires the use of a complex view to process transactions, then resolve the view into a temporary table for your program to run against. Each join or view that needs to materialize for each transaction consumes system resources and affects performance. In this approach, the system applies the join or view once (during the filtering process), populates the temporary table with the necessary information that the program needs to complete the transaction, and then runs the program against the temporary table as needed.

For example, consider the following situation:

A program needs to update 10,000 rows on the Customer table, which contains 100,000 rows of data. The Customer table is keyed by setID. To complete the transaction, the program references data that resides on a related table called PS_SET_CNTRL_REC. PS_SET_CNTRL_REC is used to associate setID and BUSINESS_UNIT values. The transaction is keyed by BUSINESS_UNIT.

Given that set of circumstances, the most efficient processing method would be similar to the following:

- Isolate affected or necessary data from both tables, and insert that into the temporary table.

Now, instead of dealing with a 10,000-row Customer table and a join to a related table, the program faces a 10,000-row temporary table that contains all of the required data to join directly to the transaction data, which can also be in a temporary table. If all necessary columns reside on the temporary tables, the program can modify all the rows at once in a simple Update statement.

This example presents two different uses of temporary tables. In one situation, the temporary table is designed to hold setup/control data in a modified form. In the other situation, the temporary table is designed to hold transaction data in a denormalized form, perhaps with additional work columns to hold intermediate calculations.

- Make sure the data appears in a denormalized form for optimum processing.
- Because the transaction is keyed by BUSINESS_UNIT, the temporary table that holds the control data should also be keyed by BUSINESS_UNIT.

In this case, the table that holds the control data is the Customer table.

Avoiding Row-by-Row Processing

A set-based program is not an all-or-nothing situation. There are some rules that call for row-by-row processing, but these rules are the exception. However, you can have a row-by-row component within a mostly set-based program.

For example, suppose your program contains five rules that you will run against your data. Four of those rules lend themselves well to a set-based approach, while the fifth requires a row-by-row process. In this situation, run the four set-based steps or rules first, and then run the row-by-row portion last to resolve the exceptions. Although not pure set-based processing, you will obtain better performance than if the entire program used a row-by-row approach.

When performing a row-by-row update, reduce the number of rows and the number of columns that you select to an absolute minimum to decrease the data transfer time.

For logic that cannot be coded entirely in set, try to process most of the transactions in set, and process only the exceptions in a row-by-row loop. A good example of an exception is the sequence numbering of detail lines within a transaction, when most transactions have only a single detail line. You can set the sequence number on all the detail lines to 1 by default, in an initial set-based operation, and then execute a Select statement to retrieve only the exceptions (duplicates) and update their sequence numbers to 2, 3, and so on.

Avoid the tendency to expand row-by-row processing for more than what is necessary. For example, just because you are touching all of the rows of a given table in a specific row-based process, you do not necessarily gain efficiency by running the rest of your logic on that table in a row-based manner.

When updating a table, you can add another column to be set in the Update statement. However, do not add another SQL statement to your loop just because your program is looping. If you can apply that SQL in a set-based manner, then, in most cases, you achieve better performance with a set-based SQL statement outside the loop.

The rest of this section describes techniques for avoiding row-by-row processing and enhancing performance.

Filtering

Using SQL, filter the set to contain only those rows that are affected or meet the criteria and then run the rule on them. Use the Where clause to minimize the number of rows to reflect only the set of affected rows.

Two-Pass Approach

Use a two-pass approach, wherein the first pass runs a rule on all of the rows, and the second pass resolves any rows that are exceptions to the rule. For instance, bypass exceptions to the rule during the first pass, and then address the exceptions individually in a row-by-row manner.

Parallel Processes

Divide sets into distinct groups, and then run the appropriate rules or logic against each set in parallel processes. For example, in terms of employee data, you could split the population into distinct sets of "hourly" and "salary," and then you could run the appropriate logic for each set in parallel.

Flat Temporary Tables

Flatten your temporary tables. The best temporary tables are denormalized and follow a flat file model for improved transaction processing.

For example, payroll control data might be keyed by set ID and effective dates rather than by business unit and accounting date. Use the temporary table to denormalize the data, and switch the keys to business unit and accounting date. Afterwards, you can construct a straight join to the Time Clock table, keyed by business unit and date.

Techniques to Avoid

Note the following:

- If you have a series of identical temporary tables, examine your refinement process.
- Don't attempt to accomplish a task that your database platform does not support, as in complex mathematics, non-standard SQL, and complex analytical modeling.

Use standard SQL for set processing.

- Although subqueries are a useful tool for refining your set, make sure that you're not using the same one multiple times.

If you are using the same subquery in more than one statement, you should probably have denormalized the query results into a temporary table. Identify the subqueries that appear frequently and, if possible, denormalize the queried data into a temporary table.

Using Set Processing Examples

Each of the following sections contains an example of set processing.

Payroll

In this example, suppose the payroll department needs to give a USD 1000 salary increase to everybody whose department made more than USD 50,000 profit. The following pseudocode enables you to compare the row-by-row and set-based approaches.

- Row-by-Row:

```
declare A cursor for select dept_id from department where profit > 50000;
open A;
fetch A into p_dept_id
while sql_status == OK
    update personnel set salary = (salary+1000) where dept_id = p_dept_id;
    fetch A into p_dept_id;
end while;
close A;
free A;
```

- Set-Based:

```
update personnel set salary = (salary + 1000)
where exists
    (select 'X' from department
     where profit > 50000
     and personnel.dept_id = department.dept_id)
```

Note. The set-based example employs a correlated subquery, which is important in set-based processing.

Temporary Tables

One technique to improve database performance is to use a temporary table to hold the results of a common subquery. Effective dating and setID indirection are common types of subqueries that you can replace with joins to temporary tables. With the joins in place, you can access the temporary table instead of doing the subquery multiple times. Not only do most databases prefer joins to subqueries, but if you combine multiple subqueries into a single join as well, the performance benefits can be significant.

In this setID indirection example, you see a join from a transaction table (keyed by BUSINESS_UNIT and ACCOUNTING_DT) to a setup table (keyed by SETID and EFFDT).

To accomplish this using a single SQL statement, you need to bring in PS_SET_CNTRL_REC to map the business unit to a corresponding setID. This is typically done in a subquery. You also need to bring in the setup table a second time in a subquery to get the effective date (MAX(EFFDT) <= ACCOUNTING_DT). If you have a series of similar statements, this can be a performance issue.

The alternative is to use a temporary table that is the equivalent of the setup table. The temporary table is keyed by BUSINESS_UNIT and ACCOUNTING_DT instead of SETID and EFFDT. You populate it initially by joining in your batch of transactions (presumably also a temporary table) once, as described previously, to get all the business units and accounting dates for this batch. From then on, your transaction and setup temporary tables have common keys, which allows a straight join with no subqueries.

For the example, the original setup table (PS_ITEM_ENTRY_TBL) is keyed by SETID, ENTRY_TYPE and EFFDT.

The denormalized temporary table version (PS_ITEM_ENTRY_TAO) is keyed by PROCESS_INSTANCE, BUSINESS_UNIT, ENTRY_TYPE and ACCOUNTING_DT, and carries the original keys (SETID and EFFDT) as simple attributes for joining to other related setup tables, as in PS_ITEM_LINES_TBL for this example.

If the program references the setup table in only one Insert/Select or Select statement, you would not see increased performance by denormalizing the temporary table. But if several SQL statements are typically executed in a single run, all of which join in the same setup table with similar setID and effective date considerations, then the cost of populating the temporary table up front provides long-term advantages.

- Original setup table version:

```

INSERT INTO PS_PG_PENDDST_TAO (...)
SELECT
. . . . .
  ( (I.ENTRY_AMT_BASE - I.VAT_AMT_BASE) * L.DST_LINE_MULTPLR * L.DST_LINE_PERCENT / =>
    100 ),

  ( (I.ENTRY_AMT - I.VAT_AMT) * L.DST_LINE_MULTPLR * L.DST_LINE_PERCENT / 100 ),
. . . . .
FROM   PS_PENDING_ITEM I, PS_PG_REQUEST_TAO R, PS_ITEM_LINES_TBL L,
       PS_ITEM_ENTRY_TBL E, PS_SET_CNTRL_REC S, PS_BUS_UNIT_TBL_AR B
. . . . .

WHERE
  AND L.ENTRY_REASON = I.ENTRY_REASON
  AND L.SETID = E.SETID
  AND L.ENTRY_TYPE = E.ENTRY_TYPE
  AND L.EFFDT = E.EFFDT
. . . . .
  AND E.EFF_STATUS = 'A'
  AND S.RECNAME = 'ITEM_ENTRY_TBL'
  AND S.SETID = E.SETID
  AND S.SETCNTRLVALUE = I.BUSINESS_UNIT
  AND E.ENTRY_TYPE = I.ENTRY_TYPE
  AND E.EFFDT = (SELECT MAX(EFFDT) FROM PS_ITEM_ENTRY_TBL Z
                  WHERE Z.SETID = E.SETID
                     AND Z.ENTRY_TYPE = E.ENTRY_TYPE
                     AND Z.EFF_STATUS = 'A'
                     AND Z.EFFDT <= I.ACCOUNTING_DT )
  AND B.BUSINESS_UNIT = I.BUSINESS_UNIT
/

```

- Denormalized temporary table version:

```

INSERT INTO PS_ITEM_ENTRY_TAO
. . . . .
SELECT DISTINCT %BIND(PROCESS_INSTANCE), I.BUSINESS_UNIT, I.ACCOUNTING_DT,
E.ENTRY_TYPE...
. . .
FROM PS_PENDING_ITEM I, PS_PG_REQUEST_TAO R,
PS_ITEM_ENTRY_TBL E, PS_SET_CNTRL_REC S, PS_BUS_UNIT_TBL_AR B
WHERE R.PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE)
AND R.PGG_GROUP_TYPE = 'B'
AND I.POSTED_FLAG = 'N'
AND R.GROUP_BU = I.GROUP_BU
AND R.GROUP_ID = I.GROUP_ID
AND E.EFF_STATUS = 'A'
AND S.RECNAME = 'ITEM_ENTRY_TBL'
AND S.SETID = E.SETID
AND S.SETCNTRLVALUE = I.BUSINESS_UNIT
AND E.ENTRY_TYPE = I.ENTRY_TYPE
AND E.EFFDT = ( SELECT MAX(EFFDT) FROM PS_ITEM_ENTRY_TBL Z
                WHERE Z.SETID = E.SETID
                  AND Z.ENTRY_TYPE = E.ENTRY_TYPE
                  AND Z.EFF_STATUS = 'A'
                  AND Z.EFFDT <= I.ACCOUNTING_DT )
AND B.BUSINESS_UNIT = I.BUSINESS_UNIT
/
INSERT INTO PS_PG_PENDDST_TAO (...)
SELECT ...
( (I.ENTRY_AMT_BASE - I.VAT_AMT_BASE) * L.DST_LINE_MULTIPLR * L.DST_LINE_PERCENT / =>
100 ),
( (I.ENTRY_AMT - I.VAT_AMT) * L.DST_LINE_MULTIPLR * L.DST_LINE_PERCENT / 100 ),

. . . . .
FROM PS_PENDING_ITEM I, PS_PG_REQUEST_TAO R, PS_ITEM_LINES_TBL L,
PS_ITEM_ENTRY_TAO E
. . . . .
WHERE
. . . . .
AND L.ENTRY_REASON = I.ENTRY_REASON
AND L.SETID = E.SETID
AND L.ENTRY_TYPE = E.ENTRY_TYPE
AND L.EFFDT = E.EFFDT
. . . . .
AND E.BUSINESS_UNIT = I.BUSINESS_UNIT
AND E.ACCOUNTING_DT = I.ACCOUNTING_DT
AND E.ENTRY_TYPE = I.ENTRY_TYPE
/

```

Platform Issues

Set processing does not behave the same on every database platform. On some platforms, set processing can encounter performance breakdowns. Some platforms do not optimize update statements that include subqueries.

For example, environments that are accustomed to updates with subqueries get all the qualifying department IDs from the Department table, and then, using an index designed by an application developer, update the Personnel table. Other platforms read through every employee row in the Personnel table and query the Department table for each row.

On platforms where these types of updates are a problem, try adding some selectivity to the outer query. In the following example, examine the SQL in the Before section, and then notice how it is modified in the After section to run smoothly on all platforms. You can use this approach to work around platforms that have difficulty with updates that include subqueries.

Note. In general, set processing capabilities vary by database platform. The performance characteristics of each database platform differ with more complex SQL and set processing constructs. Some database platforms allow additional set processing constructs that enable you to process even more data in a set-based manner. If performance needs improvement, you must tailor or tune the SQL for your environment. You should be familiar with the capabilities and limitations of your database platform and can recognize, through tracing and performance results, the types of modifications you need to incorporate with the basic set processing constructs described.

- Basic version:

```
UPDATE PS_REQ_LINE
SET SOURCE_STATUS = 'I'
WHERE
EXISTS
(SELECT 'X' FROM PS_PO_ITM_STG STG
WHERE
STG.PROCESS_INSTANCE =%BIND(PROCESS_INSTANCE) AND
STG.PROCESS_INSTANCE =PS_REQ_LINE.PROCESS_INSTANCE AND
STG.STAGE_STATUS = 'I' AND
STG.BUSINESS_UNIT = PS_REQ_LINE.BUSINESS_UNIT AND
STG.REQ_ID = PS_REQ_LINE.REQ_ID AND
STG.REQ_LINE_NBR = PS_REQ_LINE.LINE_NBR)
```

- Optimized for platform compatibility:

```
UPDATE PS_REQ_LINE
SET SOURCE_STATUS = 'I'
WHERE
PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE) AND
EXISTS
(SELECT 'X' FROM PS_PO_ITM_STG STG
WHERE
STG.PROCESS_INSTANCE =%BIND(PROCESS_INSTANCE) AND
STG.PROCESS_INSTANCE =PS_REQ_LINE.PROCESS_INSTANCE AND
STG.STAGE_STATUS = 'I' AND
STG.BUSINESS_UNIT = PS_REQ_LINE.BUSINESS_UNIT AND
STG.REQ_ID = PS_REQ_LINE.REQ_ID AND
STG.REQ_LINE_NBR = PS_REQ_LINE.LINE_NBR)
```

Note. This example assumes that the transaction table (PS_REQ_LINE) has a PROCESS_INSTANCE column to lock rows that are in process. This is another example of designing your database with batch performance and set processing in mind.

This modification enables the system to limit its scan through PS_REQ_LINE to only those rows that the program is currently processing. At the same time, it enables a more set-friendly environment to first scan the smaller staging table and then update the larger outer table.

Chapter 5

Using Meta-SQL and PeopleCode

This chapter provides an overview of Application Engine meta-Structured Query Language (SQL) and discusses how to:

- Use PeopleCode in Application Engine programs.
- Include dynamic SQL.
- Use Application Engine meta-SQL.

Understanding Application Engine Meta-SQL

Application Engine meta-SQL is divided into the following categories:

- **Construct**
A construct is a direct substitution of a value, which helps to build or modify a SQL statement.
- **Function**
A function performs an action on its own or causes another function to be called.
- **Meta-variable**
A meta-variable allows substitution of text within SQL statements.

Note. Some meta-SQL elements can be used only in Application Engine programs, some can be used both in Application Engine programs and in other environments, and some cannot be used in Application Engine programs at all. Only meta-SQL elements that can be used in Application Engine are discussed in this PeopleBook. You can find a complete reference to all PeopleSoft meta-SQL elements in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

Chapter 5, "Using Meta-SQL and PeopleCode," Application Engine Meta-SQL Reference, page 74

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," Understanding Meta-SQL

Using PeopleCode in Application Engine Programs

This section provides an overview of PeopleCode and Application Engine programs and discusses how to:

- Decide when to use PeopleCode.
- Consider the program environment.
- Access state records with PeopleCode.
- Use If/Then logic.
- Use PeopleCode in loops.
- Use the AERSection class.
- Make synchronous online calls to Application Engine programs.
- Use the file class.
- Call COBOL modules.
- Call PeopleTools APIs.
- Use the CommitWork function.
- Call WINWORD Mail Merge
- Use PeopleCode examples.

Understanding PeopleCode and Application Engine Programs

Inserting PeopleCode within Application Engine programs enables you to reuse common function libraries and improve performance. In many cases, a small PeopleCode program used instead of Application Engine PeopleCode is an excellent way to build dynamic SQL, perform simple If/Else edits, set defaults, and perform other tasks that do not require a trip to the database.

Scope of Variables

This table presents the different types of variables typically used in Application Engine programs and their scope:

Type of Variable	Scope	Comments
State record (work record)	Transaction (unit of work)	Using a work record as your Application Engine state record means that the values in the work record cannot be committed to the database. Commits happen as directed, but any values in work records are not retained after a commit.
State record (database record)	Application Engine program	Using a database record as your Application Engine state record preserves the values in the state record on commit, and the committed values are available in the event of a restart.
Local PeopleCode variables	PeopleCode program	Local PeopleCode variables are available only for the duration of the PeopleCode program that is using them.
Global PeopleCode variables	Application Engine program	Global PeopleCode variables are available during the life of the program that is currently running. Any global PeopleCode variables are saved when an Application Engine program commits and checks points; therefore, they are available in the event of a restart.
Component PeopleCode variables	Application Engine program	Component PeopleCode variables act like global variables to Application Engine.

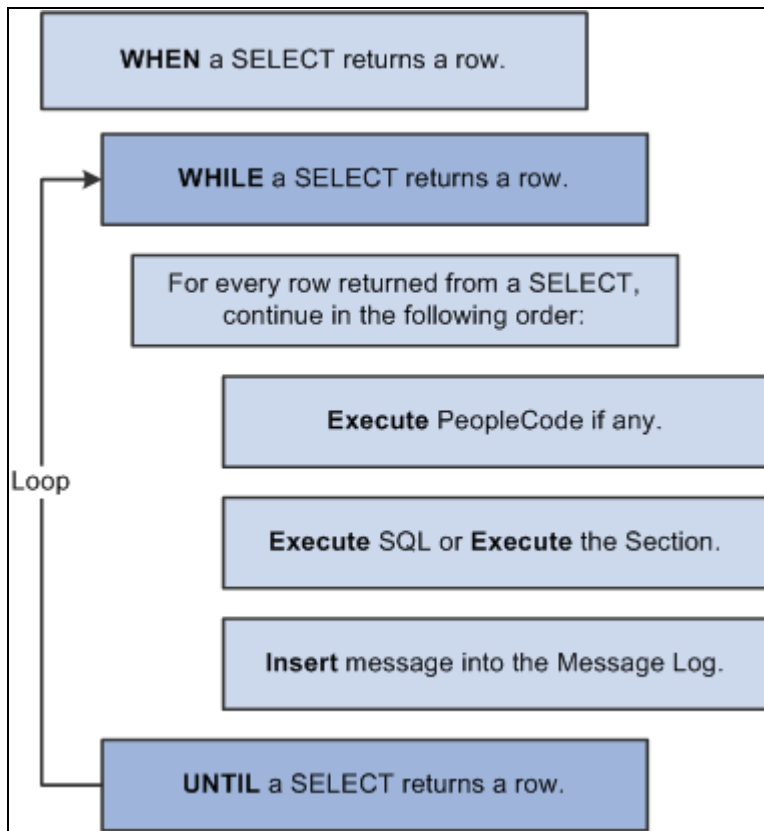
Action Execution Order

No other types of actions are required within a step in conjunction with a PeopleCode action (or program). So, you can have a step that contains nothing but one PeopleCode action. If you include other actions with your PeopleCode action within the same step, it's important to keep in mind the execution hierarchy.

With PeopleCode actions, Application Engine executes the PeopleCode program *before* the SQL, Call Section, or Log Message actions, but a PeopleCode program executes *after* any program flow checks.

Because there are multiple action types, they must execute in agreement within a system, and therefore the order in which action's execute is significant. At runtime, actions defined for a given step are evaluated based on their action type. All of the action types exist within a strict hierarchy of execution. For example, if both a Do When and PeopleCode action exist within a given step, the Do When is always executed first.

The following example shows the sequence and level of execution for each type of action:



Example of action execution hierarchy

Deciding When to Use PeopleCode

Application Engine is not intended to execute programs that include nothing but PeopleCode actions. The primary purpose of Application Engine is to run SQL against your data.

For the most part, use PeopleCode for setting If, Then, Else logic constructs, performing data preparation tasks, and building dynamic portions of SQL statements, while still relying on SQL to complete the bulk of the actual program processing. You also use PeopleCode to reuse online logic that is already developed. Also, PeopleCode is the tool for taking advantage of the new technologies, such as component interfaces and application classes.

Most programs need to check that a certain condition is true prior to executing a particular section. For example, if the hourly wage is less than or equal to X, do Step A; if not, fetch the next row. In certain instances, you need to modify variables that exist in a state record. PeopleCode enables you to set state record variables dynamically.

Avoid rowset processing in an Application Engine program. Loading data into a rowset can use a significant amount of memory, approximated by the following formula:

$$\text{mem} = \text{nrows} * (\text{row overhead} + \text{nrecords} * (\text{rec overhead} + \text{nfields} * (\text{field overhead}) + \text{average cumulative fielddata}))$$

where

- mem is the amount of memory required to store the rowset.

- `nrows` is the number of rows.
- `row overhead` is the overhead per row.
- `nrecords` is the number of records per row.
- `rec overhead` is the record overhead (approximately 40 bytes).
- `nfields` is the number of fields in the record.
- `field overhead` is the overhead per field (approximately 80 bytes).
- `average cumulative fielddata` is the average amount of data per field.

Using this formula, a rowset containing 500,000 rows with one record per row, 50 fields, and 200 bytes per field would require approximately 2.3 gigabytes of memory.

Considering the Program Environment

When writing or referencing PeopleCode in a PeopleCode action, you must consider the environment in which the Application Engine program runs. *Environment* indicates the differences between online and batch modes. Application Engine programs usually run in batch mode, and, consequently, your PeopleCode cannot access pages or controls as it can while running in online mode. Any PeopleCode operations that manipulate pages will not run successfully. Even if you invoke your Application Engine program online from a record or a page using the `CallAppEngine` PeopleCode function, the Application Engine PeopleCode still does not have direct access to component buffers.

Any record field references that appear in a PeopleCode action can refer only to fields that exist on an Application Engine state record. Component buffers, controls, and so on are still inaccessible even if you define the page records as state records on the Program Properties dialog box. An Application Engine program can access only state records or other objects you create in PeopleCode.

However, you do have several options for passing data from a component buffer to an Application Engine program. You can use the `CallAppEngine` PeopleCode function, or you can define global variables.

Passing Parameters Through the CallAppEngine Function

For individual page fields and simple PeopleCode variables, such as numbers and strings, you can use the `CallAppEngine` PeopleCode function to pass values as parameters.

To use the `CallAppEngine` function:

1. Declare a record object in PeopleCode.

Here is an example:

```
Local Record &MyRecord;
```

2. Assign the record objects to any state record that you want to pass to the Application Engine program.

Record objects are parameters to the `CallAppEngine` function.

3. Set the appropriate values on that state record.
4. Include the record object in the function call.

After these values get set in the state record, all the actions in a particular program can use the values, not just the PeopleCode actions.

Defining Global Variables

You can also define global variables or objects in PeopleCode before calling an Application Engine program. Application Engine PeopleCode actions only are able to access the variables you define; however, the PeopleCode could set a state record field equal to a number or string variable for use by other Application Engine actions.

Also, an Application Engine PeopleCode program can read or update a scroll area or a grid using a global rowset object. When accessing a scroll area or a grid from Application Engine PeopleCode, the same rules apply, and the same illegal operations are possible that you see with accessing PeopleCode not in an Application Engine program.

The parameters submitted in a CallAppEngine are by value. These parameters seed the specified Application Engine state record field with the corresponding value. If that value is changed within Application Engine by updating the state record field, the component data is not be affected. The only way to update component buffers or external PeopleCode variables from Application Engine is to use global PeopleCode variables and objects.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "PeopleCode Built-in Functions," CallAppEngine

Accessing State Records with PeopleCode

Executing PeopleCode from Application Engine steps enables you to complete some simple operations without having to use SQL. For example, to assign a literal value to an Application Engine state record field using SQL you may have issued a statement similar to the following:

```
%SELECT (MY_AET.MY_COLUMN)
SELECT 'BUSINESS_UNIT' FROM PS_INSTALLATION
```

You can use a PeopleCode assignment instead:

```
MY_AET.MY_COLUMN = "BUSINESS_UNIT" ;
```

Similarly, you can use a PeopleCode If statement instead of using a Do When action to check the value of a state record field.

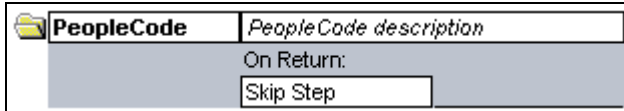
When accessing state records with PeopleCode, keep the following in mind:

- State records are unique to Application Engine programs.
- Within Application Engine PeopleCode, state record values can be accessed and modified using the standard `recordname.fieldname` notation.

Note. When you launch an Application Engine program from PeopleSoft Process Scheduler, you can generate a process warning status on completion of the program by including and modifying the AE_APPSTATUS field in a state record. You can generate the warning status by setting AE_APPSTATUS to a value of 1.

Using If/Then Logic

From PeopleCode, you can trigger an error status, or false return, by using the Exit function. Use the On Return value on the PeopleCode action properties to specify how your Application Engine program behaves according to the return of your PeopleCode program. This example shows the On Return property:



Example of On Return action property

By default, the program terminates, similar to what happens when a SQL error occurs. But by changing the On Return value to *Skip Step*, you can control the flow of your Application Engine program.

You can use Exit to add an If condition to a step or a section break. For example:

```
If StateRec.Field1 = 'N'
Exit(1);
Else
/* Do processing */
End-if;
```

You must specify a non-zero return value to trigger the On Return action. The concepts of "return 1" and "return True" are equivalent. So, if the return value is non-zero or True, then Application Engine performs what you specify for On Return, as in *Abort* or *Skip Step*. However, if the program returns zero or False, Application Engine ignores the selected On Return value.

Using PeopleCode in Loops

You can insert PeopleCode inside of a Do loop, but take care when using PeopleCode inside of high-volume Do loops (While, Select, Until). Keep the number of distinct programs inside the loop to a minimum. You should avoid having PeopleCode performing the actual work of the program and instead use it primarily to control the flow (If, Then logic), build dynamic SQL, or interact with external systems.

Using bind variables instead of literals to pass values to SQL statements is essential in PeopleCode loops or if the PeopleCode is called in a loop. If the PeopleCode loops, there is a good probability that Application Engine will use a dedicated cursor, which saves the overhead of recompiling the SQL for all iterations. If the PeopleCode is called from within a loop, Application Engine does not reduce the number of compiles, but Application Engine avoids flooding the SQL cache (for those database servers that support SQL cache) when it uses bind variables. Do not use bind variables for values in a Select list or for SQL identifiers, such as table and column names, as some databases do not support this.

Note. Null bind values of type DateTime, Date, or Time are always resolved into literals.

On those database platforms for which PeopleSoft has implemented this feature, Setting BulkMode to True often results in significant performance gains when inserting rows into a table in a loop.

In general, avoid PeopleCode calls within a loop. If you can call the PeopleCode outside of the loop, use that approach. This can increase overall performance.

Using the AERSection Class

The AERSection PeopleCode class enables you to change the properties of an Application Engine program section dynamically, without having to modify any of the Application Engine tables directly. This enables you to develop rule-based applications that conform dynamically to variables that a user submits through a page, such as the Application Engine Request page.

The AERSection class provides the following flexibility:

- Portions of SQL are determined by checks prior to execution.
- The logic flow conforms as rules change, and the program adjusts to the rules.

When using an AERSection object, keep the following in mind:

- Check to make sure that you primarily require dynamic capabilities with the SQL your program generates.
- Make sure that the rules to which your program conform are relatively static, or at least defined well enough such that a standard template could easily accommodate them.
- Consider using SQL definitions to create dynamic SQL for your programs to avoid the complexity created by the AERSection object using the StoreSQL function.
- The AERSection class is designed to dynamically update only SQL-based actions, not PeopleCode, Call Section, or other actions.

You can add a PeopleCode action to your generated section, but you can not alter the PeopleCode.

- The AERSection class is designed for use for online processing.

Typically, dynamic sections should be constructed in response to a user action.

Note. Do not call an AERSection object from an Application Engine PeopleCode action.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode API Reference, "AERSection Class"

Making Synchronous Online Calls to Application Engine Programs

To make synchronous online calls to an Application Engine program, use the PeopleCode function CallAppEngine.

Note. If you make a synchronous call, the user can't perform another PeopleSoft task until the Application Engine program completes. Consider the size and performance of the Application Engine program called by CallAppEngine. You should ensure that the program will run to successful completion consistently within an acceptable amount of time.

If an Application Engine program called by CallAppEngine terminates abnormally, the user receives an error, similar to other save time errors, that forces the user to cancel the operation. The CallAppEngine function returns a value based on the result of the Application Engine call. If the program was successful, it returns a zero, and if the program was unsuccessful, it returns a non-zero.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "PeopleCode Built-in Functions," CallAppEngine

Using the File Class

The file layout class enables you to perform file input and output operations with Application Engine using PeopleCode. A file object enables you to open a file (for reading or writing), read data from a file, or write data to it. Using the combination of the file class and Application Engine provides an effective method to integrate (or exchange) the data stored in a legacy system with your PeopleSoft system. The file class facilitates the creation of a flat file that both your legacy system and Application Engine programs support.

An Application Engine program running on the application server uses a file object to read the file sent from the legacy system and translate it, so that the file can update the affected PeopleSoft application tables. For the PeopleSoft system and the legacy system to communicate, you must first construct a file object that is compatible for both systems to insert and read data.

Attain rowset and record access for a file using a file layout definition. You create the file layout definition in Application Designer, and it acts as a template for the file that both systems read from and write to. This simplifies reading, writing, and manipulating complex transaction data with PeopleCode.

Generally, use the file class and Application Engine combination when you cannot implement the PeopleSoft Integration Broker solution.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode API Reference, "File Class"

Calling COBOL Modules

Using the PeopleCode RemoteCall function, you can call COBOL modules from a PeopleCode action. This option supports existing Application Engine programs that call COBOL modules. You can also use it to upgrade Application Engine programs from previous releases.

PTPECOBL Program

The PTPECOBL interface program is a PeopleSoft executable that enables you to invoke your called COBOL module and pass it required values. You code the RemoteCall function to invoke PTPECOBL, which in turn calls the specified COBOL module.

If you use PTPECOBL, you don't have to write your own executable to process this task. However, PTPECOBL does not perform any SQL processing other than retrieve a list of state record values. Consequently, if your current logic requires prior SQL processing, you may want to write your own executable file to call your COBOL module. In most situations, PTPECOBL saves you from having to write a custom executable file to handle each call to a generated dynamically loadable code (.GNT) file.

PTPECOBL performs the following tasks:

1. Initializes the specified state record in memory.
2. Invokes the COBOL module specified in your PeopleCode.
3. Submits required parameters to the called COBOL module.
4. Updates the state record as necessary, issues a commit, and then disconnects from the database after your program completes.

Note. While your COBOL program runs, it can access and return values to the state record.

Shared Values in Application Engine and COBOL

Note the following options for sharing values between the Application Engine program and your called COBOL program:

- Use state records.

If you add field names, Application Engine enables you to pass state record values to the called COBOL program and to get changes passed back to the calling PeopleCode program. If you pass the state record values in this manner, use PTPECACH to retrieve and update values just as PTPEFCNV does.

- Code custom SQL.

If you do not pass the initial values using state record fields, you need to insert the appropriate SQL in your called COBOL module to retrieve the appropriate values. Then, to return any updated values to the calling Application Engine program, you must insert the appropriate SQL into a PeopleCode program.

If your COBOL program needs values that do not appear in a state record field, then you can pass PeopleCode variables and values. These variables and values are then retrieved and updated by calling PTPNETRT from within your COBOL program.

- Create a custom executable file.

If you include extra SQL processing and use non-state record values, for consistency purposes, it might be a better approach to create a custom executable file. This way, you can call your program directly and have it perform all the PTPNETRT processing. Remember that a RemoteCall command can only call an executable program, not a GNT file.

Syntax and Parameters

This example shows a sample RemoteCall function, issued from an Application Engine PeopleCode action to a COBOL module:


```
RemoteCall ("PSRCCBL", ?
  "PSCOBOLPROG", "PTPECOBL", ?
  "AECOBOLPROG", "MY_GNT", ?
  "STATERECORD", "MY_AET", ?
  "PRCSINST", MY_AET.PROCESS_INSTANCE, ?
  "RETCODE", &RC, ?
  "ERRMSG", &ERR_MSG, ?
  "FIELD1", MY_AET.FIELD1, ?
  "FIELD2", MY_AET.FIELD2);
```

This table describes each parameter:

Parameters	Description
PSRCCBL	The Remote Call dispatcher. It executes the specified COBOL program using the connect information of the current operator.
PSCOBOLPROG	Specify the name of the COBOL program to run. In this case, it is PTPECOBL. This parameter makes the remote call from Application Engine distinct from a normal remote call. When you enter this parameter, you in effect enable the following parameters, some of which are required.
AECOBOLPROG	Specify the name of the COBOL module you're calling; for example, MY_GNT.
STATERECORD	Specify the appropriate state record that your Application Engine program will share with your COBOL module; for example, MY_AET. PTPECOBL then reserves space in memory for all of the fields on the state record, regardless of whether they will ultimately store values for processing.
PRCSINST	Specify the state record and Process Instance field; for example, MY_AET.PROCESS_INSTANCE. This retrieves the current process instance value that appears on the state record and submits it to your COBOL module using PTPECOBL.
RETCODE and ERRMSG	(Optional) Include RETCODE if you need to return information about any potential problems that the COBOL processing encountered, or use it if your Application Engine program must know whether it completed successfully.
Fieldnames and Values	This is where you specify any fields on the state record that contain initial values for your COBOL module. The quoted field names you specify must exist on the specified state record. The corresponding value can be a PeopleCode variable, a record.field reference, or a hard-coded value.

Commit and RemoteCall

Note the following using RemoteCall and an Application Engine program:

- The called COBOL module executes as a separate unit of work.
- Execute a commit in the step immediately preceding the step containing the RemoteCall PeopleCode action and also in the step containing the Remote Call PeopleCode action.

This enables the COBOL process to recognize the data changes made up to the point that it was called, and it also minimizes the time when the process might be in a non-restartable state.

- If you insert SQL processing into your COBOL module, commit updates are made by your module.

PTPECOBL does not issue any commits.

- If the intent of your COBOL process is to update the value of a passed state record field, then the calling Application Engine PeopleCode is responsible for ensuring that the state record field has been modified, and the Application Engine program is responsible for committing the state record updates.
- Consider how your COBOL module will react in the event of a restart.

Because the work in COBOL will have already been completed and committed, will your module ignore a duplicate call or be able to undo or redo the work multiple times? This is similar to issues faced when you execute a remote call from PeopleCode.

- Typically, when a COBOL program updates the database and then disconnects or terminates without having issuing an explicit commit or rollback, an implicit rollback occurs.

Without an explicit commit, the database does not retain any updates.

Note. By default, RemoteCall does not generate any log files after the program completes. To generate and retain the .out and .err log files, you must set the *RCCBL Redirect* parameter in the PeopleSoft Process Scheduler configuration file to a value of *1*.

See *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*, "PeopleCode Built-in Functions," SetNextPanel.

See Also

Enterprise PeopleTools 8.51 PeopleBook: PeopleSoft Process Scheduler, "Using the PSADMIN Utility," Editing the PeopleSoft Process Scheduler Configuration File

Calling PeopleTools APIs

You can call all of the PeopleTools APIs from an Application Engine program. Keep the following items in mind when using APIs:

- All the PeopleTools APIs contain a Save method.

However, when you call an API from your Application Engine program, regardless of the API's Save method, the data does not actually get saved until the Application Engine program issues a commit.

- If you've called a component interface from an Application Engine program, all the errors related to the API get logged in the PSMessages collection associated with the current session object.
- If you've sent a message, the errors get written to the message log and the Application Engine message log.
- If an Application Engine program called from message subscription PeopleCode encounters errors and the program exits (with `Exit (1)`), the error is written to the message log and is marked as an error

Using the CommitWork Function

This function commits pending changes (inserts, updates, and deletes) to the database. Keep the following in mind when using CommitWork:

- This function applies only to a batch Application Engine program.
If the program is invoked by CallAppEngine, the CommitWork function is ignored. The same is true for commit settings at the section or step level.
- This function can be used only in an Application Engine program that has restart disabled.
- The CommitWork function is useful only when you are doing row-at-a-time SQL processing in a single PeopleCode program, and you need to commit without exiting the program.

In a typical Application Engine program, SQL commands are split between multiple Application Engine actions that fetch, insert, update, or delete application data. You use the section or step level commit settings to manage the commits.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "PeopleCode Built-in Functions," CommitWork

Calling WINWORD Mail Merge

If the Process Scheduler is booted using a shared drive on another machine, and you intend to call a WINWORD mail merge process from Application Engine, one of the following must be done to ensure successful completion:

1. Configure the Process Scheduler to run Application Engine programs using psae instead of psaesrv.
2. Ensure the generated document is saved locally, and not on a shared network drive.

Using PeopleCode Examples

The following sections provide some examples of common ways that you can utilize PeopleCode within Application Engine programs.

Do When Actions

Instead of a Do When action that checks a %BIND value, you can use PeopleCode to perform the equivalent operation. For example, suppose the following SQL exists in your program:

```
%SELECT(EXISTS) SELECT 'Y' FROM PS_INSTALLATION WHERE %BIND(TYPE) = 'X' ,
```

Using PeopleCode, you could insert the following code:

```

If TYPE = 'X' Then
    Exit(0);
Else
    Exit(1);
End-if;

```

If you set the On Return parameter on the PeopleCode action properties to *Skip Step*, this code behaves the same as the Do When action. The advantage of using PeopleCode is that there is no trip to the database.

Dynamic SQL

If you have a Select statement that populates a text field with dynamic SQL, such as the following:

```

%SELECT(AE_WHERE1)
SELECT 'AND ACCOUNTING_DT  <= %Bind(ASOF_DATE) '

```

You can use this PeopleCode:

```

AE_WHERE1 = "AND ACCOUNTING_DT  <= %Bind(ASOF_DATE) ";

```

Sequence Numbering

If you typically use Select statements to increment a sequence number inside of a Do Select, While, or Until loop, you can use the following PeopleCode instead:

```

SEQ_NBR = SEQ_NBR + 1;

```

Using PeopleCode rather than SQL is significant. Because the sequencing task occurs repeatedly inside a loop, the cost of using a SQL statement to increment the counter increases with the volume of transactions your program processes. When you are modifying a program to take advantage of PeopleCode, the areas of logic you should consider are those that start with steps that are executed inside a loop.

Note. You can also use the meta-SQL constructs %Next and %Previous when performing sequence numbering. Using these constructs may help performance in both PeopleCode and SQL calls.

Rowsets

You can use rowsets in Application Engine PeopleCode. However, using rowsets means you'll be using PeopleCode to handle more complicated processing, which degrades performance.

Math Functions

Use the math functions that your database offers whenever possible.

Internally, PeopleCode assigns types to numeric values. Calculations for the Decimal type are processed in arrays to ensure decimal point uniformity across hardware and operating system environments. This is much slower than calculations for type Integer, which are processed at the hardware level.

When PeopleCode converts strings to numeric values, it does so using the internal Decimal type. For performance reasons, avoid calculations using these values.

A third type of numeric value is the Float type. It is not used as frequently, for the following reasons:

- Constants are never stored as Float types in the compiled code.

For example, 2.5 is always Decimal.

- The only way to produce a Float value is by using built-in functions, such as Float or the Financial math functions.

The Float type is used to produce a float result only if all operands are also of the Float type. Float operations occur at the hardware level.

PeopleCode does not offer optimum performance when processing non-Integer, non-Float math calculations. To perform calculations with these numeric types, consider allowing the database to perform the calculations in COBOL.

PeopleCode supports a range of mathematical functions and numeric types. Generally speaking, if a complex calculation is executed repetitively in an Application Engine program, careful analysis should be done to determine whether to perform the calculation in a PeopleCode action or to use the relational database management (RDBMS) functions through a SQL action. Using SQL may require PeopleSoft meta-SQL to handle platform differences, but it may be the most efficient way to update field values. If SQL is not appropriate, consider numeric typing in PeopleCode, as this affects the speed and accuracy of the calculation.

SQL Class

Instead of using the SQL class within PeopleCode, have Application Engine issue the SQL and use a Do Select action that loops around sections containing PeopleCode actions.

It might appear easier to code all of the logic within a single PeopleCode program, but splitting the logic into smaller pieces is preferable because you will have better performance, and you get a finer granularity of commit control. Within a PeopleCode program, you can commit in certain cases using the CommitWork function. You can always issue a commit between Application Engine steps.

See *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*, "PeopleCode Built-in Functions," CommitWork.

See *PeopleTools 8.51 PeopleBook: PeopleCode API Reference*, "SQL Class," Understanding SQL Objects and Application Engine Programs.

Arrays

Instead of using arrays in Application Engine PeopleCode, explore the use of temporary tables for storing pertinent or affected data. This has the following advantages:

- Data is available for restarts.
- An RDBMS is efficient at managing and searching tables.
- Using temporary tables also lends itself to set-based processing.

You can use the Statement Timings and PeopleCode Detail Timings trace options to generate an Application Engine timings report to determine whether your program is spending significant time processing arrays.

Including Dynamic SQL

Typically, developers include dynamic constructs in Application Engine programs to change SQL based on various runtime factors or on user-defined input entered through a page. You can include dynamic SQL in Application Engine programs in a variety of ways. For example, you can use:

- Dynamic sections, using the AERSection object.
- Changing SQL, using the SQL class.
- References to SQL in your own tables.

The AERSection class is primarily designed for online section building, and therefore won't be the most frequently used solution.

Use the SQL class to store SQL in a SQL definition that you can also access in Application Designer. Then, if you have a few SQL statements to execute, generate the SQL IDs based on some methodology, such as a timestamp, and then store these in a table.

When the program runs, your SQL could query this table based on process and extract the appropriate SQL IDs to be executed with a SQL action in a Do Select loop.

```
%SQL( %BIND( MY_SQLID, NOQUOTES ) )
```

For a dynamic Do action, the AE_APPLID and the AE_SECTION fields must appear on the default state record.

Application Engine Meta-SQL Reference

This section describes the meta-SQL constructs, functions, and meta-variables you can use in Application Engine.

Note. The SQL Editor does not validate all of the meta-SQL constructs, such as %Bind and %Select. Messages might appear stating these constructs are invalid.

%Abs

Description

Because the %Abs function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Abs

%AeProgram

Description

Use the %AeProgram meta-variable to specify a quoted string containing the currently executing Application Engine program name.

%AeSection

Description

Use the %AeSection meta-variable to specify a quoted string containing the currently executing Application Engine section name.

%AeStep

Description

Use the %AeStep meta-variable to specify a quoted string containing the currently executing Application Engine Step name.

%AsOfDate

Description

Use the %AsOfDate meta-variable to specify a quoted string containing the as of date used for the current process.

%AsOfDateOvr

Description

Use the %AsOfDateOvr meta-variable only as a parameter of the %ExecuteEdits function, to override the default use of the system date with the value of a field on a joined record.

See Also

Chapter 5, "Using Meta-SQL and PeopleCode," %Table, page 107

%BINARYSORT

Description

Because the %BINARYSORT construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %BINARYSORT

%Bind

Syntax

```
%Bind([recordname.]fieldname [, NOQUOTES][, NOWRAP][, STATIC])
```

Description

Use the %Bind construct to retrieve a field value from a state record. Can be used anywhere in a SQL statement. When executed, %Bind returns the value of the state record field identified within its parentheses.

Notes About %Bind

Typically, when you use %Bind to provide a value for a field or a Where condition, the type of field in the state record that you reference with %Bind must match the field type of the corresponding database field used in the SQL statement.

On most platforms, you cannot use a literal to populate a Long Varchar field. You should use the %Bind (*recordname.fieldname*) construct.

In the case of an external call to a section in another program, if the called program has its own default state record defined, then Application Engine uses that default state record to resolve the %Bind(*fieldname*). Otherwise, the called program inherits the default state record of the calling program.

All fields referenced by a %Select construct must be defined in the associated state record.

You must use the Date, Time, and DateTime output wrappers in the Select list that populates the state record fields.

This ensures compatibility across all supported database platforms. For example:

- First SQL action:

```
%Select (date_end)
  SELECT %DateOut (date_end )
    FROM PS_EXAMPLE
```


- Second SQL action:

```
INSERT INTO PS_EXAMPLE
VALUES(%Bind(date_end))
```

Bind Variables and Date Wraps

The behavior of bind variables within Application Engine PeopleCode and normal PeopleCode is the same.

Alternately, if you compare Application Engine SQL to PeopleCode (of any type), then the system processes bind variables differently.

If you use the following approach:

```
AND TL_EMPL_DATA1.EFFDT <= %P(1))
```

Then in PeopleCode you issue:

```
%SQL(MY_SQL, %DateIn(:1))
```

This assumes that you have referenced the literal as a bind variable.

Or in Application Engine SQL you issue:

```
%SQL(MY_SQL, %Bind(date_field))
%SQL(MY_SQL, %Bind(date_field, NOWRAP))
```

Parameters

<i>Parameter</i>	<i>Description</i>
<i>Recordname</i>	The name of a state record. If you do not specify a particular state record, Application Engine uses the default state record to resolve the %Bind (<i>fieldname</i>).
<i>Fieldname</i>	The field defined in the state record.
NOQUOTES	If the field specified is a character field, its value is automatically enclosed in quotes unless you use the NOQUOTES parameter. Use NOQUOTES to include a dynamic table and field name reference, even an entire SQL statement or clause, in an Application Engine SQL action.

Parameter	Description
NOWRAP	<p>If the field is of type Date, Time, or DateTime, the system automatically wraps its value in %DateIn or %DateOut, unless you use the NOWRAP parameter. Therefore, if the state record field is populated correctly, you do not need to be concerned with the inbound references, although you can suppress the inbound wrapping with the NOWRAP modifier inside the %Bind. Furthermore, Application Engine skips the inbound wrapper if the %Bind (<i>date</i>) is in the select field list of another %Select statement. This is because the bind value is already in the outbound format, and the system selects it into another state record field in memory. In this circumstance there is no need for either an outbound wrapper or an inbound wrapper. For example,</p> <p>First SQL action:</p> <pre>%Select(date_end) SELECT %DateOut(date_end) FROM PS_GREG</pre> <p>Second SQL action:</p> <pre>INSERT INTO ps_greg VALUES(%Bind(date_end))</pre>
STATIC	<p>The STATIC parameter enables you to include a hard-coded value in a reused statement. For %Bind instances that contain dynamic SQL, this parameter must be used in conjunction with the NOQUOTES parameter for proper execution of a reused statement.</p>

Example

```
UPDATE PS_REQ_HDR
  SET IN_PROCESS_FLG = %Bind(MY_AET.IN_PROCESS_FLG) ,
      PROCESS_INSTANCE = %Bind(PROCESS_INSTANCE)
 WHERE IN_PROCESS_FLG = 'N'
    AND BUSINESS_UNIT || REQ_ID
    IN (SELECT BUSINESS_UNIT || REQ_ID
        FROM PS_PO_REQRCON_WK1
        WHERE PROCESS_INSTANCE = %Bind(PROCESS_INSTANCE))
```

In the previous example, %Bind (PROCESS_INSTANCE) assigns the value of the field PROCESS_INSTANCE in the default state record to the PROCESS_INSTANCE field in table PS_REQ_HDR.

The %Bind construct is also used in a Where clause to identify rows in the table PS_PO_REQRCON_WK1, in which the value of PROCESS_INSTANCE equals the value of PROCESS_INSTANCE in the default state record.

%Cast

Description

Because the %Cast function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Cast

%ClearCursor**Syntax**

```
%ClearCursor( { program,section,step, action | ALL } )
```

Description

Use the %ClearCursor function to recompile a reused statement and reset any STATIC %Bind variables.

When you use the %ClearCursor function, keep the following in mind:

- The function must be located at the beginning of the statement.
- %ClearCursor can be the only function or command contained in the statement.

Parameters

<i>Parameter</i>	<i>Description</i>
<i>program</i>	Specify the name of the Application Engine program containing the reused statement you want to recompile.
<i>section</i>	Specify the name of the section containing the reused statement you want to recompile.
<i>step</i>	Specify the name of the step containing the reused statement you want to recompile.
<i>action</i>	Specify one of the following values: <ul style="list-style-type: none"> • D: Do Select. • H: Do When. • N: Do Until. • W: Do While. • S: SQL.
ALL	Clear all cursors in the current Application Engine program.

%COALESCE

Description

Because the %COALESCE function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %COALESCE

%Comma

Description

Use the %Comma meta-variable to specify a comma. This is useful where you must use a comma, but commas are not allowed due to the parsing rules. For example, you might use this if you wanted to pass a comma, as a parameter, to the %SQL meta-SQL function.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %SQL

%Concat

Description

Because the %Concat meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Concat

%CurrentDateIn

Description

Because the %CurrentDateIn meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %CurrentDateIn

%CurrentDateOut

Description

Because the %CurrentDateOut meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %CurrentDateOut

%CurrentDateTimeIn

Description

Because the %CurrentDateTimeIn meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %CurrentDateTimeIn

%CurrentDateTimeOut

Description

Because the %CurrentDateTimeOut meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements,"
%CurrentDateTimeOut

%CurrentTimeIn

Description

Because the %CurrentTimeIn meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %CurrentTimeIn

%CurrentTimeOut

Description

Because the %CurrentTimeOut meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %CurrentTimeOut

%DateAdd

Description

Because the %DateAdd function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateAdd

%DateDiff

Description

Because the %DateDiff function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateDiff

%DateIn

Description

Because the %DateIn construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateIn

%DateNull

Description

Because the %DateNull meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateNull

%DateOut**Description**

Because the %DateOut function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateOut

%DatePart**Description**

Because the %DatePart function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DatePart

%DateTimeDiff**Description**

Because the %DateTimeDiff function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateTimeDiff

%DateTimeIn

Description

Because the %DateTimeIn construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateTimeIn

%DateTimeNull

Description

Because the %DateTimeNull meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateTimeNull

%DateTimeOut

Description

Because the %DateTimeOut function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DateTimeOut

%DecDiv

Description

Because the %DecDiv function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DecDiv

%DecMult**Description**

Because the %DecMult function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DecMult

%DTTM**Description**

Because the %DTTM function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %DTTM

%EffDtCheck**Description**

Because the %EffDtCheck construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %EffDtCheck

%Execute

Syntax

```
%Execute( [ / ] )
command1{ ; | / }
command2{ ; | / } ...
commandN{ ; | / }
```

Description

Use the %Execute function to execute database-specific commands from within your Application Engine program. Also, the %Execute function enables you to include multiple statements in a single Application Engine action without encountering database-specific differences. For instance, there are instances where you could code a single Application Engine action to contain multiple SQL statements, and they may run successfully on one database platform. However, if you attempt to run the same code against a different database platform, you might encounter errors or skipped SQL.

By default, Application Engine expects a semicolon to be used to delimit multiple commands within an %Execute function statement. You can instruct Application Engine to use a forward slash (/) delimiter instead by placing a forward slash inside the function parentheses.

Note. When you use the %Execute function, it must be located at the beginning of the statement and can be the only function or command contained in the statement. The action type must be SQL.

Example

The following code enables you to use an Oracle PL/SQL block in an %Execute statement:

```
%Execute( / )
DECLARE
  counter INTEGER;
BEGIN
  FOR counter := 1 TO 10
    UPDATE pslock SET version = version + 1;
  END FOR;
END;
/
```

%ExecuteEdits

Syntax

```
%ExecuteEdits(type,recordname [alias][, field1,field2, ...])
```

Description

Use the %ExecuteEdits function to apply data dictionary edits in batch. The %ExecuteEdits function is Application-Engine-only meta-SQL. You cannot use it in COBOL, SQR, or PeopleCode—not even in Application Engine PeopleCode.

Notes About %ExecuteEdits

Note the following:

- Consider performance carefully when using this function.

Prompt table and Translate table edits have a significant impact, because they involve correlated subqueries. Run a SQL trace at execution time so that you can view the SQL generated by %ExecuteEdits. Look for opportunities where it can be optimized.

- In general, %ExecuteEdits is best used on a temporary table.

If you must run this against a real application table, you should provide Where clause conditions to limit the number of rows to include only those that the program is currently processing. Process the rows in the current set all at once rather than processing them row by row.

- With %ExecuteEdits, you cannot use work records in a batch, set-based operation.

All higher-order key fields used by prompt table edits must exist on the record that your code intends to edit, and the field names must match exactly. For example,

```
%ExecuteEdits(%Edit_PromptTable, MY_DATA_TMP)
```

The record MY_DATA_TMP contains the field STATE with a prompt table edit against PS_REGION_VW, which has key fields COUNTRY and REGION. The REGION field corresponds to STATE, and COUNTRY is the higher-order key. For %ExecuteEdits to work correctly, the MY_DATA_TMP record must contain a field called COUNTRY. It is permissible for the edited field (STATE) to use a different name, because Application Engine always references the last key field (ignoring EFFDT).

- In Application Engine, %ExecuteEdits uses the system date when performing comparisons with effective date (EFFDT); however, in some cases, this is not appropriate (Journal Edit, Journal Import, and so on). In these situations, Journal Date should be used when comparing with EFFDT. To override a program's use of the default system date with a selected field from a joined table, use %AsOfDateOvr. For example,

```
%ExecuteEdits(%AsOfDateOvr(alias.fieldname), %Bind(...))
```

- Restrict the number and type of edits to the minimum required.

Do not edit fields that are known to be valid or that are given default values later in the process. Also, consider using a separate record with edits defined specifically for batch, or provide a list of fields to be edited.

Parameters

<i>Parameter</i>	<i>Description</i>
<i>type</i>	Specify any combination of the following (added together): <ul style="list-style-type: none"> • %Edit_Required • %Edit_YesNo • %Edit_DateRange • %Edit_PromptTable • %Edit_TranslateTable
<i>recordname</i>	Specify the record used to obtain the data dictionary edits.
<i>field1, field2, ...</i>	Specify a subset of the record's fields to which edits apply.

Example

Suppose you want to insert rows with missing or invalid values in three specific fields, selecting data from a temporary table but using edits defined on the original application table. Notice the use of an alias, or correlation name, inside the meta-SQL.

```
INSERT INTO PS_JRNL_LINE_ERROR (...)
SELECT ... FROM PS_JRNL_LINE_TMP A
WHERE A.PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE)
      AND %EXECEDEEDITS(%Edit_Required + %Edit_PromptTable, ?
JRNL_LINE A, BUSINESS_UNIT, JOURNAL_ID, ACCOUNTING_DT)
```

To update rows in a temporary table that have some kind of edit error, you can use custom edits defined on the temporary table record:

```
UPDATE PS_PENDITEM_TAO
SELECT ERROR_FLAG = 'Y'
WHERE PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE)
      AND %EXECEDEEDITS(%Edit_Required + %Edit_YesNo + %Edit_DateRange + ?
%Edit_PromptTable + %Edit_TranslateTable, PENDITEM_TAO)
```

%FirstRows

Description

Because the %FirstRows meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %FirstRows

%GetProgText**Syntax**

```
%GetProgText( &Prog, &Section, &Market, &Platform, &Effdt, &Step, &Event )
```

Description

The %GetProgText function returns a string with the text of a PeopleCode program uniquely identified by the parameters.

Parameters

<i>Parameter</i>	<i>Description</i>
<i>&Prog</i>	A string with the name of an Application Engine program.
<i>&Section</i>	A string with the name of an Application Engine program section.
<i>&Market</i>	A string specifying the market for an Application Engine program section.
<i>&Platform</i>	A string specifying the platform for an Application Engine program section.
<i>&Effdt</i>	A string specifying the effective date for an Application Engine program section.
<i>&Step</i>	A string specifying a step in an Application Engine program section.
<i>&Event</i>	A string specifying the PeopleCode event.

Returns

A string containing the text of a PeopleCode program.

Example

```
&PeopleCodeText = GetProgText( "DYNROLE_PUBL", "MAIN", "GBL", "default",  
"1900-01-01", "Step03", "OnExecute" );
```

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Developer's Guide, "Using the SQL Editor"

%InsertSelect**Description**

Because the %InsertSelect construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %InsertSelect

%InsertSelectWithLongs**Description**

Because the %InsertSelectWithLongs construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements,"
%InsertSelectWithLongs

%JobInstance**Description**

Use the %JobInstance meta-variable to specify the numeric (unquoted) PeopleSoft Process Scheduler job instance.

%Join**Description**

Because the %Join construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Join

%LeftParen**Description**

Use the %LeftParen meta-variable to specify a left parenthesis. Usage is similar to %Comma.

See Also

Chapter 5, "Using Meta-SQL and PeopleCode," %Comma, page 80

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %SQL

%Like**Description**

Because the %Like construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Like

%LikeExact**Description**

Because the %LikeExact construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %LikeExact

%List

Syntax

```
%List ({FIELD_LIST | FIELD_LIST_NOLONGS | KEY_FIELDS | ORDER_BY}, recordname [correlation_id])
```

Description

The %List construct expands into a list of field names, delimited by commas. The fields included in the expanded list depends on the parameters.

Note. This meta-SQL is not implemented for COBOL, dynamic view SQL, or PeopleCode.

Considerations Using %List

When using %List in an Insert/Select or Insert/Values or %Select statement, you must have matching pairs of %List (or %ListBind) variables in the target and source field lists, using the same list type argument and record name to ensure consistency.

Parameters

<i>Parameter</i>	<i>Description</i>
FIELD_LIST	Use all field names in the given record. You can select only one option from FIELD_LIST, ORDER_BY, FIELD_LIST_NOLONGS, or KEY_FIELDS.
KEY_FIELDS	Use all key fields in the given record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS, KEY_FIELDS, or ORDER_BY.
ORDER_BY	Use all the key fields of <i>recordname</i> , adding the DESC field for descending key columns. This parameter is often used when the list being generated is for an Order By clause. You can select only one option from FIELD_LIST, KEY_FIELDS, ORDER_BY, or FIELD_LIST_NOLONGS.
FIELD_LIST_NOLONGS	Use all field names in the given record, except any long columns (long text or image fields.) You can select only one option from FIELD_LIST, ORDER_BY, KEY_FIELDS, or FIELD_LIST_NOLONGS.
<i>recordname</i>	Identify either a record or a subrecord that the field names are drawn from. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You cannot specify RECORD. <i>recname</i> , a record name in quotation marks, or a table name.
<i>correlation_id</i>	Identify the single-letter correlation ID to relate the record specified by <i>recordname</i> and its fields.

Example

The following is a good example of using %List. Both the Insert and Select statements use the same %List variable.

```
INSERT INTO PS_PO_DISTRIB_STG ( %Sql(POCOMMONDISTSTGFLDLSTU)
, %List(FIELD_LIST, CF16_AN_SBR)
, MERCHANDISE_AMT
, MERCH_AMT_BSE
, QTY_DEMAND
, QTY_PO
, QTY_PO_STD
, QTY_REQ)
SELECT %Sql(POCOMMONDISTSTGFLDLSTU)
, %List(FIELD_LIST, CF16_AN_SBR)
, MERCHANDISE_AMT
, MERCH_AMT_BSE
, QTY_DEMAND
, QTY_PO
, QTY_PO_STD
, QTY_REQ
FROM PS_PO_DIST_STG_WRK WRK
WHERE WRK.PROCESS_INSTANCE = %Bind(PROCESS_INSTANCE)
```

The following example shows a poor example of how to use %List. The Insert and Select field lists both use %List, but the Select field list is only partly dynamic. The rest is hard-coded.

```
INSERT INTO PS_EN_TRN_CMP_TMP (%List(FIELD_LIST, EN_TRN_CMP_TMP))
SELECT B.EIP_CTL_ID
, %List(SELECT_LIST, EN_BOM_COMPS A)
, E.COPY_DIRECTION
, E.BUSINESS_UNIT_TO
, E.BOM_TRANSFER_STAT
, 'N'
, B.MASS_MAINT_CODE
, 0
FROM PS_EN_BOM_COMPS A
, PS_EN_ASSY_TRN_TMP B
, PS_EN_TRNS_TMP E
WHERE ...
```

The following example shows the previous poor example rewritten in a better way:

```

INSERT INTO PS_EN_TRN_CMP_TMP (EIP_CTL_ID,
, %List(FIELD_LIST, EN_BOM_COMPS)
, COPY_DIRECTION
, BUSINESS_UNIT_TO
, BOM_TRANSFER_STAT
, EN_MMC_UPDATE_FLG
, MASS_MAINT_CODE
, EN_MMC_SEQ_FLG01
, ...
, EN_MMC_SEQ_FLG20)
SELECT B.EIP_CTL_ID
, %List(FIELD_LIST, EN_BOM_COMPS A)
, E.COPY_DIRECTION
, E.BUSINESS_UNIT_TO
, E.BOM_TRANSFER_STAT
, 'N'
, B.MASS_MAINT_CODE
, 0
, ...
, 0
FROM PS_EN_BOM_COMPS A
, PS_EN_ASSY_TRN_TMP B
, PS_EN_TRNS_TMP E
WHERE ...

```

The following code segment is another poor example. Only the field list of the Insert statement is dynamically generated, and the Select statement is statically coded. If the table STL_NET_TBL is reordered, the Insert statement will be incorrect.

```

INSERT INTO PS_STL_NET_TBL (%List(FIELD_LIST, STL_NET_TBL ) )
SELECT :1
, :2
, :3
, :4
, :5
, :6
, :7
, :8
FROM PS_INSTALLATION

```

The following code shows the previous poor example rewritten in a better way:

```

INSERT INTO PS_STL_NET_TBL (%List(FIELD_LIST, STL_NET_TBL))
VALUES (%List(BIND_LIST, STL_NET_TBL MY_AET))

```

%ListBind

Syntax

```

%ListBind({FIELD_LIST | FIELD_LIST_NOLONGS | KEY_FIELDS}, recordname [
State_record_alias])

```

Description

The %ListBind meta-SQL construct expands a field list as bind references for use in an Insert/Value statement.

Note. This meta-SQL is not implemented for COBOL, dynamic view SQL, or PeopleCode.

Considerations Using %ListBind

When using %ListBind in an insert/select or insert/values or %Select statement, you must have matching pairs of %List or %ListBind in the target and source field lists, using the same list type argument and record name to ensure consistency.

Parameters

<i>Parameter</i>	<i>Description</i>
FIELD_LIST	Use all field names in a record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS, or KEY_FIELDS.
FIELD_LIST_NOLONGS	Use all field names in a record, except any long columns (long text or image fields). You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS, or KEY_FIELDS.
KEY_FIELDS	Use all key field names in a record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS, or KEY_FIELDS.
<i>recordname</i>	Identify either a record or a subrecord that the field names are drawn from. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You cannot specify RECORD. <i>recname</i> , a record name in quotation marks, or a table name.
<i>State_record_alias</i>	Specify the Application Engine state record buffer that contains the values (this could be different than the record used to derive the field list). If missing, the default state record is assumed.

Example

```
INSERT INTO PS_TARGET (FIELD1, FIELD2, %List(FIELD_LIST, CF_SUBREC), FIELDN)⇒
VALUES (%Bind(MY_AET.FIELD1), %Bind(MY_AET.FIELD2), %ListBind(FIELD_LIST, CF_⇒
SUBREC MY_AET), %Bind(MY_AET.FIELDN))
```

%ListEqual

Syntax

```
%ListEqual({ALL | KEY }, Recordname [alias], RecordBuffer [, Separator])
```

Description

The %ListEqual construct maps each field, possibly to an alias with a %Bind value, with a separator added before each equality. Each field is mapped as follows:

```
alias.X = %Bind(recbuffer.X)
```

This construct can be used in the Set clause of an Update statement or in a Where clause.

Note. This meta-SQL is not implemented for COBOL, dynamic view SQL, or PeopleCode.

Parameters

<i>Parameter</i>	<i>Description</i>
ALL KEY	Specify if you want all fields or just key fields.
<i>recordname</i>	Identify either a record or a subrecord that the field names are drawn from. This can be a bind variable, a record object, or a record name in the form <i>recname</i> . You cannot specify RECORD. <i>recname</i> , a record name in quotation marks, or a table name.
<i>alias</i>	(Optional) Specify an alias to precede each field name.
<i>RecordBuffer</i>	Specify the record buffer for the bind variables (this could be different than the record used to derive the field list).
<i>Separator</i>	If you want to specify a logical separator, specify either AND or OR with this parameter. If you do not specify a separator, no logical separator is used; the value of a comma is used instead.

Example

```
UPDATE PS_TEMP
SET %ListEqual(ALL, CF_SUBREC, MY_AET)
WHERE %ListEqual(KEYS, TEMP, MY_AET, AND)
```

%Mod

Description

Because the %Mod function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Mod

%Next and %Previous

Description

Use the %Next and %Previous functions to return the value of the next or previous field in a numbered sequence. These functions are valid in any Application Engine SQL action, and should be used when performing sequence-numbering processing. Typically, you use them in place of a %Bind construct. These functions use the current value of the number field as a bind variable, and then increment (%Next) or decrement (%Previous) the value after the statement is executed successfully. A number field indicates the numeric field on the state record that you have initially set to a particular value (as in 1 to start).

If the statement is a Select and no rows are returned, the field value is not changed. The substitution rules are the same as for %Bind. For example, if the ReUse property is enabled, then the field is a true bind (':n' substituted). Otherwise, inline substitution occurs.

Example

You could use these functions in an Update statement within a Do Select action.

- Do Select action

```
%SELECT(field1, field2, ...) SELECT key1, key2, ... FROM PS_TABLE WHERE ...  
ORDER BY key1, key2, ..."
```

- SQL

```
UPDATE PS_TABLE SET SEQ_NBR = %Next(seq_field) WHERE key1 = %Bind(field1)  
AND key2 = %Bind(field2) ...
```

With a Do Select action, the increment/decrement occurs once per execution, not once for every fetch. So unless your Do Select action implements the Reselect property, the value is changed only on the first iteration of the loop. Alternatively, with the Reselect property or Do While and Do Until actions, every iteration re-executes the Select statement and then fetches one row. With these types of loops, the value changes on every iteration.

See Also

[Chapter 5, "Using Meta-SQL and PeopleCode," %Bind, page 76](#)

%NoUpperCase

Description

Because the %NoUpperCase construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %NoUppercase

%NumToChar**Description**

Because the %NumToChar construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %NumToChar

%ProcessInstance**Description**

Use the %ProcessInstance meta-variable to specify the numeric (unquoted) process instance.

%ResolveMetaSQL**Syntax**

```
%ResolveMetaSQL( &SQL )
```

Description

The %DatabaseRelease function returns a string with any meta-SQL in the string expanded to platform-specific SQL, similar to the text that is returned on the Meta-SQL tab when using the Resolve Meta-SQL functionality in the SQL Editor.

If &SQL does not contain any meta-SQL then the function returns a string identical to &SQL.

Parameters

<i>Parameter</i>	<i>Description</i>
<i>&SQL</i>	Specify a string containing the SQL to be resolved.

Returns

A string with meta-SQL expanded to platform-specific SQL.

Example

Here is an example:

```
&SQLText = FetchSQL(SQL.PTLT_CODE_MARKET);
&ResolveSQLText = ResolveMetaSQL(&SQLText);
```

Suppose &SQLText contains the following SQL:

```
INSERT INTO %Table(PTLT_ASSGN_TASK) (PTLT_FEATURE_CODE
, PTLT_TASK_CODE
, PORTAL_NAME
, PTLT_TASK_CODE2
, MENUNAME
, OBJECTOWNERID)
SELECT A.EOLT_FEATURE_CODE
, %Sql(PTLT_TASK_CODE, A.PNLGRPNAME, A.MARKET)
, 'EMPLOYEE'
, %Sql(PTLT_TASK_CODE, A.PNLGRPNAME, A.MARKET)
, A.MENUNAME
, ' '
FROM %Table(EOLT_FEAT_COMP) A
, %Table(PTLT_TASK) B
, %Table(PTLT_TASK_LOAD) C
WHERE %Sql(PTLT_TASK_CODE, A.PNLGRPNAME, A.MARKET) = B.PTLT_TASK_CODE
AND B.PTLT_TASK_CODE = C.PTLT_TASK_CODE
AND B.PTLT_LOAD_METHOD = C.PTLT_LOAD_METHOD
AND A.MENUNAME <> ' '
AND A.MENUNAME <> C.MENUNAME
AND NOT EXISTS (
SELECT 'X'
FROM %Table(PTLT_ASSGN_TASK) Z
WHERE Z.PTLT_FEATURE_CODE = A.EOLT_FEATURE_CODE
AND Z.PTLT_TASK_CODE = %Sql(PTLT_TASK_CODE, A.PNLGRPNAME, A.MARKET))
```

&ResolveSQLText would contain the following text (depending on your database platform):


```

INSERT INTO PS_PTLT_ASSGN_TASK (PTLT_FEATURE_CODE
, PTLT_TASK_CODE
, PORTAL_NAME
, PTLT_TASK_CODE2
, MENUNAME
, OBJECTOWNERID)
SELECT A.EOLT_FEATURE_CODE
, RTRIM(SUBSTR( A.PNLGRPNAME
,1
,18)) || ' ' || A.MARKET
, 'EMPLOYEE'
, RTRIM(SUBSTR( A.PNLGRPNAME
,1
,18)) || ' ' || A.MARKET
, A.MENUNAME
, ' '
FROM PS_EOLT_FEAT_COMP A
, PS_PTLT_TASK B
, PS_PTLT_TASK_LOAD C
WHERE RTRIM(SUBSTR( A.PNLGRPNAME,1,18)) || ' ' || A.MARKET = B.PTLT_TASK_CODE
AND B.PTLT_TASK_CODE = C.PTLT_TASK_CODE
AND B.PTLT_LOAD_METHOD = C.PTLT_LOAD_METHOD
AND A.MENUNAME <> ' '
AND A.MENUNAME <> C.MENUNAME
AND NOT EXISTS (
SELECT 'X'
FROM PS_PTLT_ASSGN_TASK Z
WHERE Z.PTLT_FEATURE_CODE = A.EOLT_FEATURE_CODE
AND Z.PTLT_TASK_CODE = RTRIM(SUBSTR( A.PNLGRPNAME,1,18)) || ' ' || A.MARKET)

```

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Developer's Guide, "Using the SQL Editor"

%ReturnCode

Description

Use the %ReturnCode meta-variable to evaluate or specify the return code of the last Application Engine program step performed. If the operation fails, breaks, or generates an error, %ReturnCode is set to one of the following types of return codes:

- Database (SQL) call errors.
- PeopleCode function errors.
- GEN_ERROR, when produced by general runtime exceptions.
- AE_ABORT, when produced by application or runtime logic, including some memory-related errors.

If the application process is not terminated, %ReturnCode is reset to the default value of 0 for each subsequent successful operation.

%RightParen

Description

Use the %RightParen meta-variable to specify a right parenthesis. Usage is similar to %Comma.

See Also

Chapter 5, "Using Meta-SQL and PeopleCode," %Comma, page 80

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %SQL

%Round

Description

Because the %Round function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Round

%RoundCurrency

Syntax

```
%RoundCurrency( expression, [ALIAS.]currency_field)
```

Description

Use the %RoundCurrency function to return the value of an amount field rounded to the currency precision specified by the field's Currency Control Field property, as defined in the Application Designer Record Field Properties dialog box. For this function to work, you must have the Multi-Currency option selected on the PeopleTools Options page.

See *PeopleTools 8.51 PeopleBook: System and Server Administration*, "Using PeopleTools Utilities," Using Administration Utilities.

This function is an enhanced version of the Application Engine &ROUND construct that appeared in previous releases, and is valid only in Application Engine SQL; it is not valid for SQLExecs or view text.

You can use this function in the Set clause of an Update statement or the Select list of an Insert/Select statement. The first parameter is an arbitrary expression of numeric values and columns from the source tables that computes the monetary amount to be rounded. The second parameter is the control currency field from a particular source table (the Update table, or a table in the From clause of an Insert/Select statement). This field identifies the corresponding currency value for the monetary amount.

Note. Remember that the as of date of the Application Engine program is used for obtaining the currency-rounding factor. The currency-rounding factor is determined by the value of DECIMAL_POSITIONS on the corresponding row in PS_CURRENCY_CD_TBL, which is an effective-dated table.

If multicurrency is not in effect, the result is rounded to the precision of the amount field (either 13.2 or 15.3 amount formats are possible).

Example

```
UPDATE PS_PENDING_DST
  SET MONETARY_AMOUNT =
    %RoundCurrency( FOREIGN_AMOUNT * CUR_EXCHNG_RT, CURRENCY_CD )
  WHERE GROUP_BU = %Bind(GROUP_BU) AND GROUP_ID = %Bind(GROUP_ID)
```

%RunControl

Description

Use the %RunControl meta-variable to specify a quoted string containing the current run control identifier. The run control ID is available to your program when using %RunControl, regardless of whether there's a row in the AEREQUEST table.

%Select

Syntax

```
%Select(statefield1[, statefield2]...[, statefieldN])
```

```
Select field1[, field2]...[, fieldN]
```

The *statefields* must be valid fields on the state record (they may be fieldname or recordname.fieldname, as with %Bind), and *fields* must be either valid fields in the From tables or hard-coded values.

Description

Use the %Select construct to identify the state record fields to hold the values returned by the corresponding Select statement. The %Select construct is required at the beginning of all Select statements. For example, you need one in the flow control actions and one in the SQL actions that contain a Select statement.

You use the %Select construct to pass variables to the state record, and you use the %Bind construct to retrieve or reference the variables.

Example

Consider the following sample statement:

```
%SELECT(BUSINESS_UNIT,CUST_ID)
SELECT BUSINESS_UNIT, CUST_ID
FROM PS_CUST_DATA
WHERE PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE)
```

The following steps illustrate the execution of the previous statement:

1. Resolve bind variables.

The string %Bind(PROCESS_INSTANCE) is replaced with the value of the state record field called PROCESS_INSTANCE.

2. Execute the SQL Select statement.
3. Perform a SQL Fetch statement.

If a row is returned, the state record fields BUSINESS_UNIT and CUST_ID are updated with the results. If the Fetch statement does not return any rows, all fields in the %Select construct retain their prior values.

Note. All fields referenced by a %Select construct must be defined in the associated state record. Also, aggregate functions always return a row, so they always cause the state record to be updated. As such, for aggregate functions, there is no difference whether you use %SelectInit or %Select.

%SelectInit

Syntax

```
%SelectInit(statefield1[, statefield2]...[, statefieldN])
Select field1[, field2]...[, fieldN]
```

The *statefields* must be valid fields on the state record (they may be *fieldname* or *recordname.fieldname*, as with %Bind), and *fields* must be either valid fields in the From tables or hard-coded values.

Description

Use the %SelectInit construct to identify the state record fields to hold the values returned by the corresponding Select statement.

The %SelectInit construct is identical to the %Select construct, with the following exception: if the Select statement returns no rows, %SelectInit reinitializes the buffers. In the case of a %Select construct where no rows are returned, the state record fields retain their previous values.

Note. For aggregate functions, there is no difference whether you use %SelectInit or %Select.

%Space

Description

Use the %Space meta-variable to specify a single space. Usage is similar to %Comma.

See Also

[Chapter 5, "Using Meta-SQL and PeopleCode," %Comma, page 80](#)

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %SQL

%SQL

Description

Use the %SQL construct to specify a SQL object, which replaces the %SQL construct in a statement. This enables commonly used SQL text to be shared among Application Engine and PeopleCode programs alike. In Application Engine, you use %Bind to specify bind variables. In PeopleCode SQL, you can use

```
:record.field
```

or

```
:1
```

If you create SQL objects that you plan to share between Application Engine and PeopleCode programs, the %SQL construct enables you to pass parameters for resolving bind variables without being concerned with the difference in the bind syntax that exists between Application Engine and PeopleCode. However, the base SQL statement that uses %SQL to represent a shared object with binds needs to be tailored to Application Engine or to PeopleCode.

When a SQL object specified has more than one version, the database type always takes precedence. That is:

- If one or more versions of a SQL definition are found for the database type of the current database connection, and if any of the versions have an effective date less than or equal to the current date, the most recent version is used.
- If no versions are found for the current database type, or if all of the versions have effective dates greater than the current date, the system looks for an effective version of the SQL definition under the database type "generic". If no version is found, an error occurs.

Example

For example, assume that your SQL is similar to the following:

```
UPDATE PS_TEMP_TBL SET ACTIVE = %BIND(MY_AET.ACTIVE)
WHERE PROCESS_INSTANCE = %ProcessInstance
```

That would not be valid if the SQL ran in PeopleCode. However, if you define your SQL as shown, you could use parameters in %SQL to insert the appropriate bind variable:

```
UPDATE PS_TEMP_TBL SET ACTIVE = %P(1)
WHERE PROCESS_INSTANCE = %ProcessInstance
```

From Application Engine, the base SQL, or source statement, might look like the following:

```
%SQL( SQL_ID, %BIND(MY_AET.ACTIVE) )
```

The PeopleCode SQL may appear as the following:

```
%SQL( SQL_ID, :MY_AET.ACTIVE )
```

Note. You can use %SQL only to reference SQL objects created directly in Application Designer. For instance, you can not use %SQL to reference SQL that resides within a section in an application library. Common SQL should be stored as a proper SQL object.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %SQL

%SQLRows

Description

Use the %SQLRows meta-variable to specify whether a SQL action returned any rows.

Can be used in any Application Engine SQL statement, but the underlying value is affected only by SQL actions. It is not affected by Do When, Do Select, Do While, and Do Until actions. For Select statements, the value can only be 0 or 1: row not found or rows found, respectively. It does not reflect the actual number of rows that meet the Where criteria. To find the number of rows that meet the Where criteria, code a Select Count (*) statement.

%Substring

Description

Because the %Substring function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Substring

%Table

Syntax

%Table(*recname*)

Description

Use the %Table construct to return the SQL table name for the record specified with *recname*.

This construct can be used to specify temporary tables for running parallel Application Engine processes across different subsets of data.

Example

For example, the following statement returns the record PS_ABSENCE_HIST:

```
%Table ( ABSENCE_HIST )
```

If the record is a temporary table and the current process has a temporary table instance number specified, then %Table resolves to that instance of the temporary table PS_ABSENCE_HIST nn , where nn is the instance number.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Table

%Test

Description

Because the %Test construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Test

%TextIn

Description

Because the %TextIn construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TextIn

%TimeAdd

Description

Because the %TimeAdd construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TimeAdd

%TimeIn

Description

Because the %TimeIn construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TimeIn

%TimeNull

Description

Because the %TimeNull meta-variable can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TimeNull

%TimeOut**Description**

Because the %TimeOut construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TimeOut

%TimePart**Description**

Because the %TimePart function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TimePart

%TrimSubstr**Description**

Because the %TrimSubstr function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TrimSubstr

%Truncate

Description

Because the %Truncate function can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Truncate

%TruncateTable

Syntax

```
%TruncateTable( table name )
```

Description

Use the %TruncateTable construct to invoke a bulk delete command on a table. It's functionally identical to a Delete SQL statement with no Where clause, but it is faster on databases that support bulk deletes. If you're familiar with COBOL, this construct is an enhanced version of the COBOL meta-SQL construct with the same name.

Some database vendors have implemented bulk delete commands that decrease the time required to delete all the rows in a table by not logging rollback data in the transaction log. For the databases that support these commands, Application Engine replaces %TruncateTable with Truncate Table SQL. For the other database types, %TruncateTable is replaced with Delete From SQL.

You should commit after the step that immediately precedes the step containing the %TruncateTable statement. In general, it is best to use this construct early in your Application Engine program as an initialization task. In addition, avoid using this meta-SQL when your Application Engine program is started from the PeopleCode CallAppEngine function.

Unlike the COBOL version, Application Engine determines if a commit is possible prior to making the substitution. If a commit is possible, Application Engine makes the substitution and then forces a checkpoint and commit after the successful execution of the delete.

If a commit is not possible, Application Engine replaces the meta-SQL with a Delete From string. This ensures restart integrity when your program runs against a database where there is an implicit commit associated with Truncate Table or where rollback data is not logged.

For databases that either execute an implicit commit for %TruncateTable or require a commit before or after this meta-SQL, replace %TruncateTable with an unconditional delete in the following circumstances:

- A commit is not allowed, as in within an Application Engine program called from PeopleCode.

- The program issues a non-select SQL statement since the last commit occurred. In such a situation, data is likely to have changed.
- You are deferring commits in a Select/Fetch loop within a restartable program.

Note. To use a record name as the argument for %TruncateTable (instead of an explicit table name), you must include a %Table meta-SQL function to resolve the unspecified table name. For example, to specify the record PO_WEEK as the argument, use the following statement:

```
%TruncateTable( %Table( PO_WEEK ) ).
```

See Also

Chapter 5, "Using Meta-SQL and PeopleCode," %Table, page 107

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %TruncateTable

%UpdateStats

Syntax

```
%UpdateStats(record name , [HIGH/LOW] )
```

For example,

```
%UpdateStats( PO_WRK1 )
```

The default is LOW.

Description

Use the %UpdateStats construct to generate a platform-dependent SQL statement that updates the system catalog tables used by the database optimizer in choosing optimal query plans. Use this construct after your program has inserted large amounts of data into a temporary table that will be deleted before the end of the program run. This saves you from having to use dummy seed data for the temporary table and having to update statistics manually.

Notes About %UpdateStats

For databases that either execute an implicit commit for %UpdateStats or require a commit before or after this meta-SQL, Application Engine skips %UpdateStats in the following circumstances:

- A commit is not allowed, as in within an Application Engine program called from PeopleCode.
- The program issues a non-select SQL statement since the last commit occurred.

In such a situation, data is likely to have changed.

- You are deferring commits in a Select/Fetch loop in a restartable program.

Application Engine skips %UpdateStats even if the previous condition is false.

The following table shows how the %UpdateStats construct is resolved by the supported database systems.

Database Function	Behavior
MSS %UpdateStats	<p>Specifying LOW produces the statement</p> <pre>UPDATE STATISTICS <i>tablename</i></pre> <p>Specifying HIGH produces the statement</p> <pre>UPDATE STATISTICS <i>tablename</i> WITH FULLSCAN</pre>
Sybase %UpdateStats	<p>LOW and HIGH = UPDATE ALL STATISTICS <i>tablename</i></p>
Oracle %UpdateStats	<p>PeopleSoft uses DDL templates (in PSDDLMODEL) to determine SQL statements for %UpdateStats. Use DDLORA.DMS to change.</p> <p>Specifying LOW produces the statement</p> <pre>execute DBMS_STATS.GATHER_TABLE_STATS (ownname=> 'PT8468908', tabname=>'PSSTATUS', estimate_percent=> >20, method_opt=> 'FOR ALL INDEXED COLUMNS SIZE=> 1',cascade=>TRUE)</pre> <p>Specifying HIGH produces the statement</p> <pre>execute DBMS_STATS.GATHER_TABLE_STATS (ownname=> 'PT848908', tabname=>'PSSTATUS', estimate_percent=> dbms_stats.auto_sample_size, method_opt=> 'FOR ALL=> INDEXED COLUMNS SIZE 1',cascade=>TRUE)</pre>
DB2 UNIX %UpdateStats	<p>In DB2 UNIX, %UpdateStats is performed by issuing sqlustat() calls that are equivalent to SQL statements. The sqlustat() is an internal DB2 API call function rather than an SQL command.</p> <p>Specifying LOW is equivalent to issuing the statement</p> <pre>RUNSTATS ON TABLE <i>tablename</i> AND INDEXES ALL</pre> <p>Specifying HIGH is equivalent to issuing the statement</p> <pre>RUNSTATS ON TABLE <i>tablename</i> WITH DISTRIBUTION AND=> DETAILED INDEXES ALL</pre> <p>Note. You cannot view the sqlustat() calls nor the RUNSTATS statement in the SQL trace.</p>

Database Function	Behavior
DB2 390 %UpdateStats	<p>Uses a DDL model template (in PSDDLMODEL) to format a control statement for the DB2 UDB for OS390 and z/OS Runstats utility. Refer to the PeopleTools Installation Guide and the Administration Guide for more details on using %UpdateStats with DB2 UDB for OS390 and z/OS.</p> <p>Specifying LOW produces the statement</p> <pre>RUNSTATS TABLESPACE [DBNAME].[TBSPCNAME] TABLE([=> DBNAME].[TABLE]) SAMPLE 25 [INDEXLIST] REPORT NO=> SHRLEVEL CHANGE UPDATE ACCESSPATH</pre> <p>Specifying HIGH produces the statement</p> <pre>RUNSTATS TABLESPACE [DBNAME].[TBSPCNAME] TABLE([=> DBNAME].[TABLE]) [INDEXLIST] REPORT NO SHRLEVEL=> CHANGE UPDATE ACCESSPATH</pre>
Informix %UpdateStats	<p>Specifying LOW produces the statement</p> <pre>UPDATE STATISTICS MEDIUM FOR TABLE tablename</pre> <p>Specifying HIGH produces the statement</p> <pre>UPDATE STATISTICS HIGH FOR TABLE tablename</pre>

%UpdateStats Database Considerations

The following table lists potential issues that you might encounter when using %UpdateStats.

Database	Consideration
Microsoft SQL Server Sybase UDB	<p>PeopleSoft forces a commit before and after the %UpdateStats statement. Therefore, the system skips this meta-SQL if a commit is not allowed. For instance, a commit is not allowed in the following situations:</p> <ul style="list-style-type: none"> • The Application Engine program is not running in batch mode. • You have issued non-Select/Fetch SQL (in which the data is likely to change) since the last commit. • You are deferring commits in a Select/Fetch loop within a restartable program.
Oracle	<p>Oracle has an implicit commit after the %UpdateStats statement executes. Same behavior as previous consideration.</p>

Database	Consideration
DB2 UDB for OS/390 and z/OS	<p>For DB2 UDB for OS/390 and z/OS, %UpdateStats requires IBM stored procedure DSNUTILS running in an authorized Work Load Manager Application Environment. It is also highly recommended that individual tables intended to be a target of the %UpdateStats function are segregated to their own tablespaces. Refer to the following documents for more details on using %UpdateStats: PeopleTools Installation Guide for DB2 UDB for OS/390 and z/OS; PeopleTools Administration Guide for DB2 UDB for OS/390 and z/OS.</p> <p>Note. You can trace information messages from the Runstats command on DB2 for z/os executed as a result of issuing %UpdateStats. To enable this trace, select the SQL Informational Trace check box on the Configuration Manager – Trace page.</p>
Informix IBM UDB	%UpdateStats locks the table being analyzed on UDB and Informix. Therefore, use this meta-SQL only on tables that are not likely to be concurrently accessed by other applications and users. You might use %UpdateStats to analyze Application Engine dedicated temporary tables.
All	%UpdateStats consumes a large amount of time and database resources if run against very large tables. Therefore, analyze permanent data tables outside of application programs. Also, if temporary tables are likely to grow very large during a batch run, run the batch program only with %UpdateStats enabled to seed the statistics data or when the data composition changes dramatically.

Disabling %UpdateStats

You can disable %UpdateStats in the following ways:

- Include the following parameter on the command line when running an Application Engine program:
`-DBFLAGS 1`
- Change the Dbflags=0 parameter in the PeopleSoft Process Scheduler configuration file (or PSADMIN) to Dbflags=1.

Using %UpdateStats With COBOL

You can use the %UpdateStats construct from SQL embedded in COBOL programs. Use this syntax:

```
%UpdateStats(tablename)
```

When you issue this construct from PeopleTools, the parameter is *record name*.

%Upper

Description

Because the %Upper construct can be used in more than just Application Engine programs, it is documented in the *PeopleTools 8.51 PeopleBook: PeopleCode Language Reference*.

See Also

PeopleTools 8.51 PeopleBook: PeopleCode Language Reference, "Meta-SQL Elements," %Upper

Chapter 6

Managing Application Engine Programs

This chapter discusses how to:

- Run Application Engine programs.
- Debug Application Engine programs.
- Restart Application Engine programs.
- Cache the Application Engine server.
- Free locked temporary tables.

Running Application Engine Programs

This section provides an overview of program execution options and discusses how to:

- Create process definitions.
- List process definition parameters.
- Start programs with the Application Engine Process Request page.
- Use PeopleCode to invoke Application Engine programs.
- Use the command line to invoke Application Engine programs.

Understanding Program Execution Options

You execute Application Engine programs in one of the following modes: batch using PeopleSoft Process Scheduler, online using a PeopleCode function, and manually using the command line. The following table lists some differences between online and batch programs:

<i>Online Execution</i>	<i>Batch Execution</i>
Started by the CallAppEngine function from PeopleCode.	Started through PeopleSoft Process Scheduler.
Program runs quickly, synchronously, and at random times.	Programs run for longer amounts of time, asynchronously, and at scheduled times.

<i>Online Execution</i>	<i>Batch Execution</i>
Potential for simultaneous execution.	Can be designed for parallel execution for performance.
Uses the online temporary table pool.	Uses the batch/dedicated temporary table pool.

Batch Programs Using PeopleSoft Process Scheduler

This is the most typical mode of execution. You invoke programs that run in this mode using PeopleSoft Process Scheduler or the Application Engine Process Request page. Batch mode is also referred to as asynchronous execution, meaning that it runs independently in the background. Application Engine runs on any operating system that PeopleSoft supports as an application server. If your site uses an operating system that is not supported for Application Engine, you must run Application Engine programs on the application server. (The only exception is OS/390 [z/OS]).

To run Application Engine programs on the batch server, you must install BEA Tuxedo. This applies to both UNIX and to Microsoft Windows NT batch servers. If you run your batch server on the same server machine as your application server, then the application server and the batch server can share one BEA Tuxedo installation. If your batch server is separate from your application server, then you must install BEA Tuxedo on your batch server.

The TOOLBINSRV parameter in the PeopleSoft Process Scheduler configuration file determines where PeopleSoft Process Scheduler invokes an Application Engine program. For high-volume batch environments, specify the PS_HOME\bin\server\winx86 directory that exists on the same machine where the Application Engine program runs.

Online Programs Using PeopleCode

Application Engine programs that execute online are typically executed from a page with the CallAppEngine PeopleCode function. Such online processes are synchronous, meaning that subsequent processes wait on the results. For instance, a page may be frozen until the online process returns the necessary results. With the CallAppEngine function, there are no Commit statements issued. However, if you use the asynchronous online PeopleCode option, ProcessRequest, Commit statements are allowed.

Manual Programs Using the Command Line

Usually, you use this technique only during testing or if you need to manually restart a program.

Creating Process Definitions

Select PeopleTools, Process Scheduler, Processes to access the Processes - Process Definition page.

Process Definition		Process Definition Options	Override Options	Destination	
Process Type:	Application Engine				
Name:	AEMINTEST				
*Description:	Simple AE test program		<input checked="" type="checkbox"/> API Aware		
Long Description:	Simple AE program to test that AE works.		<input type="checkbox"/> Restart Enabled?		
			Retry Count: 0		
*Priority:	Medium		Retention Days:	0	
*Process Category:	Default		Default Category		
System Constraints					
Max Concurrent:			Max Processing Time:		minutes
Mutually Exclusive Process(es)					
Customize Find First 1 of 1 Last					
	*Process Type	*Process Name	Description		
1				+	-

Processes - Process Definition page

Process Definition		Process Definition Options	Override Options	Destination	
Process Type:	Application Engine				
Name:	AEMINTEST				
*Description:	Simple AE test program		<input checked="" type="checkbox"/> API Aware		
Long Description:	Simple AE program to test that AE works.		<input type="checkbox"/> Restart Enabled?		
			<input type="checkbox"/> Read Only		
*Priority:	Medium		Retention Days:	0	Retry Count: 0
*Process Category:	Default		Default Category		
System Constraints					
Max Concurrent:			Max Processing Time:		minutes
Mutually Exclusive Process(es)					
Customize Find First 1 of 1 Last					
	*Process Type	*Process Name	Description		
1				+	-

Processes - Process Definition page

To use PeopleSoft Process Scheduler for starting Application Engine batch programs, create a process definition for each program. Running Application Engine programs is very similar to running any COBOL or Structured Query Report (SQR) program that you typically invoke with PeopleSoft Process Scheduler. Use Application Engine as the generic process type definition. Each Application Engine program that you invoke using PeopleSoft Process Scheduler requires a unique process definition derived from the generic process type definition.

Note. When creating a process definition based on the Application Engine process type definition, the process name you assign must exactly match your Application Engine program name.

Listing Process Definition Parameters

Select PeopleTools, Process Scheduler, Processes, Process Definition Options to access the Processes - Process Definition Options page.

The screenshot displays the 'Processes - Process Definition Options' page. At the top, there are five tabs: 'Process Definition', 'Process Definition Options' (which is the active tab), 'Override Options', 'Destination', and 'Page Transfer'. Below the tabs, the 'Process Type' is set to 'Application Engine' and the 'Name' is 'AEMINTEST'. There are input fields for 'Server Name' and 'Recurrence Name', each with a magnifying glass icon for search. The 'On File Creation' section includes a 'File Dependency' checkbox, a 'Wait for File' input field, and a 'Time Out Max Minutes' input field. The 'System Recovery Process' section has 'Process Type' and 'Process Name' input fields, both with search icons. The 'Process Security' section contains two lists: 'Component' with 'AE_REQUEST' and 'PRCSMULTI' (each with a search icon and +/- buttons), and 'Process Groups' with 'TLSALL' (with a search icon and +/- buttons).

Processes - Process Definition Options page

Use this page to list parameters. The complete parameter list is:

- -ct MICROSOFT
- -cd %%DBNAME%%
- -co %%OPRID%%
- -cp %%OPRPSWD%%
- -r %%RUNCNTLID%%
- -i %%INSTANCE%%
- -ai %%PRCSNAME%

Starting Programs with the Application Engine Process Request Page

You can also start an Application Engine program by using the Application Engine Process Request page. Using this request page enables you to specify additional values and parameters than those that appear within PeopleSoft Process Scheduler process definitions.

Most users start Application Engine programs from an application-specific request page using PeopleSoft Process Scheduler. A systems expert or power user may, at times, need to create custom process requests that require multiple programs to perform parallel processing or that need to set specific, initial values in a state record. This is an example of where you might use the Application Engine process request page.

Note. Generally, if seed data or other Application Engine request settings are required for a particular program, the application-specific request page has SQL executables that do the work transparent to the user. Typically, no user should invoke programs from the generic process request page. Use this page for internal testing and as a basis for designing program-specific request pages.

Tables Used in the Process Request Page

The Application Engine process request page inserts values into the following tables:

- **AEREQUESTTBL.**
Contains all of the values that appear on the page except those in the Parameters group.
- **AEREQUESTPARM.**
Includes only initial state record values specified in the Parameters group, if needed.

Note. Inserting a row in either of the Application Engine request tables is not required to run an Application Engine program. This is a key difference from versions of Application Engine prior to PeopleTools 8, where a row in Application Engine request tables is required to start a program regardless of how it is invoked. The run control ID is available to your program using %RunControl, whether or not a row is inserted into the AEREQUESTTBL table.

You need to use the Application Engine Request page to invoke Application Engine and insert a row into the Application Engine request records only if you need to perform any of the following tasks:

- Insert initial values into the state records associated with a particular program.
- Set an as of date for the Application Engine program to perform retroactive processing.
- Set a non-default market for the program.
- Set up a temporary table image to use if you are submitting a PeopleSoft EPM process request that performs parallel processing. Refer to PeopleSoft EPM application documentation for details.

Note. Entries in the AEREQUESTTBL table do not have any effect on Application Engine programs called from PeopleCode using the CallAppEngine function.

Application Engine Requests

Select PeopleTools, Application Engine, Request AE to access the Application Engine Request page.

Application Engine Request

User ID:PTDOCAORun Control ID:PORTAL_MTUPG

Program Name:PORTAL_MENUPortal Menu Import

Run

Last Run

Process Origin:Other

Process Instance:

Status:Pending

Process Frequency:Once

Market:

As Of Date:

Parameters

State Record:

*Bind Variable Name:

Value:

Date:

Application Engine Request page

Process Origin	Displays where the program was invoked: from PeopleSoft Process Scheduler, from the command line, and so on.
Process Instance	Displays the process instance assigned to the previous program run.
Status	Displays the status of the last program run, whether it is successful, pending, and so on.
Process Frequency	<div>Specify how long a particular process request will remain active or valid:</div> <ul style="list-style-type: none"><i>Always</i>: Select to run the process request as needed.<i>Once</i>: Select if a process request is a one-time-only request.<i>Don't</i>: Select to disable a process request, so that no one invokes it and potentially corrupts data.
As Of Date	If you are requesting retroactive processing, specify the appropriate as of date.
Bind Variable Name	Enter the appropriate field or bind variable for which you are inserting a value.
Value	Enter the initial value that you want to set for the specified field.

Using PeopleCode to Invoke Application Engine Programs

To call a particular Application Engine program from a page using PeopleCode, use the CallAppEngine function in SavePreChange or SavePostChange PeopleCode. The basic syntax for CallAppEngine is as follows:

```
CallAppEngine(applid [, statereclist ]);
```

Note. The RemoteCall function is no longer valid for invoking Application Engine programs in PeopleCode. However, the RemoteCall function still applies to calling other COBOL functions. If you don't convert the RemoteCall PeopleCode that previously called an Application Engine program to use the new function, an error message appears.

Use CallAppEngine if the program you are invoking is a quick process. Because the process is synchronous, a user must wait for any process invoked by CallAppEngine to complete before doing anything else. If the called program causes an unreasonable delay, then use another alternative, such as the ScheduleProcess PeopleCode function.

Use CallAppEngine when you have a complex, SQL-intensive business process that must run in batch and online, or the process requires the use of dedicated temporary tables. If this is not the case, you are usually better off writing the entire program in native PeopleCode. If you've written logic in PeopleCode, presumably for online execution, and you want to reuse it in a batch program, you may be forced into row-by-row processing. Design the batch logic first, and then decide whether to have a separate online version or just reuse the batch code using CallAppEngine. Consider the trade-off between code reuse and performance. It is inherently more difficult, but not impossible, to develop a common solution that performs adequately in both batch and online environments.

Do not use CallAppEngine within an Application Engine PeopleCode step. If you need to call an Application Engine program from another Application Engine program, you must use the Call Section action.

Do not use CallAppEngine to control the commit operation. Programs called with CallAppEngine are embedded within a larger unit of work defined by the page trigger, such as a page save.

Note. Online PeopleCode that calls CallAppEngine should be set to run on the application server. You encounter performance issues if you run PeopleCode on the client in a three-tier configuration, because every SQL statement that Application Engine issues must be serialized and then sent to the application server for execution.

Using the Command Line to Invoke Application Engine Programs

You might invoke an Application Engine program through the command line in the following situations:

- Restarting.

When a program abends, a system administrator might restart the program using the command line. If needed, you can locate all of the specific program and process information from Process Monitor on the Process Request Detail dialog box. Normally, users (or system administrators) perform a restart from Process Monitor.

- Development or Testing.

Many developers include the command line in a batch file to launch a program they are developing or testing. This way, they can quickly execute the batch file as needed. This also enables separation of development of the application program from its associated pages.

- Debugging.

To debug a program running on the server, you can sign into the server (using telnet, for example) and invoke the program from the command line.

To start an Application Engine program from the command line, you must specify the Application Engine executable (PSAE.EXE) followed by the required parameters, as shown in the following example:

```
psae -CT dbtype -CS server -CD database_name -CO oprid -CP oprpswd?  
-R run_control_id -AI program_id -I process_instance -DEBUG (Y|N)?  
-DR (Y|N) -TRACE tracevalue -DBFLAGS flagsvalue -TOOLSTRACESQL value?  
-TOOLSTRACEPC value -OT outtype -OF outformat -FP filepath
```

Or, if your command line options are stored in a text file, you can enter:

```
psae optfilename
```

Note. For Microsoft Windows NT and UNIX servers, you must set the PS_SERVER_CFG environment variable before you invoke an Application Engine program from the command line. PS_SERVER_CFG must contain the fully qualified name of a correctly configured Process Scheduler PSPRCS.CFG file. When Application Engine runs from the command line, it resolves %PS_SERVDIR% to the value of the environment variable PS_SERVDIR instead of the parent directory of a Process Scheduler configuration.

Command Line Options

-CT	Specify the type of database to which you are connecting. Values are <i>ORACLE</i> , <i>MICROSFT</i> , <i>SYBASE</i> , <i>INFORMIX</i> , <i>DB2UNIX</i> , and <i>DB2ODBC</i> .
-CS	Required for Sybase and Informix. For platforms that require a server name as part of sign-on, enter the appropriate server name. This affects Sybase, Informix, and Microsoft SQL Server. However, for Microsoft SQL Server, this option is valid but not required.
-CD	Enter the name of the database to which the program will connect.
-CO	Enter the user ID of the person who is running the program.
-CP	Enter the password associated with the specified user ID.
-R	Enter the run control ID to use for this run of the program.
-AI	Specify the Application Engine program to run.
-I	Required for restart. Enter the process instance for the program run. The default is 0, which means Application Engine uses the next available process instance.
-DEBUG	This parameter controls the Debug utility. Enter <i>Y</i> to indicate that you want the program to run in debugging mode, or enter <i>N</i> to indicate that you do not.
-DR	This parameter controls restart disabling. Enter <i>Y</i> to disable restart, or enter <i>N</i> to enable restart.

-TRACE

To enable tracing from the command line, enter this parameter and a specific trace value. The value you enter is the sum of the specific traces that you want to enable. Traces and values are:

1: Initiates the Application Engine step trace.

2: Initiates the Application Engine SQL trace.

128: Initiates the Application Engine timings file trace, which is similar to the COBOL timings trace.

256: Includes the PeopleCode detail timings in the 128 trace.

1024: Initiates the Application Engine timings table trace, which stores the results in database tables.

2048: Initiates the database optimizer explain, writing the results to the trace file. This option is supported only on Oracle, Informix, and Microsoft SQL Server.

4096: Initiates the database optimizer explain, storing the results in the Explain Plan table of the current database. This option is supported only on Oracle, DB2, and Microsoft SQL Server.

For example, to enable the 1, 2, and 128 traces, you would enter *131*, the sum of 1, 2, and 128. To indicate that you do not want any traces, enter *0*. If you do not explicitly enter *0*, Application Engine uses the trace value set in PeopleSoft Configuration Manager.

8192: Sets a trace for PeopleSoft Integration Broker transform programs.

16384: Initiates the statement timings trace, but stores the results in the following tables: PS_AE_TIMINGS_LG and PS_AE_TIMINGS_DT.

-DBFLAGS

To disable %UpdateStats meta-SQL construct, enter *1*.

-TOOLSTRACESQL

Enable the SQL trace.

-TOOLSTRACEPC

Enable the PeopleCode trace.

-OT

(Optional) Initialize the PeopleCode meta-variable %OutDestType (numeric).

PeopleCode example of %OutDestType:

```
&ProcessRqst.OutDestType = %OutDestType ;
```

-OF

(Optional) Initialize the PeopleCode meta-variable %OutDestFormat (numeric).

PeopleCode example of %OutDestFormat:

```
Query.RunToFile(Record QryPromptRecord, %OutDestFormat);
```

-FP

(Optional) Initialize the PeopleCode meta-variable %FilePath (string).

PeopleCode example of %FilePath:

```
If All(%FilePath) Then
    &FILENAME = %FilePath | &FILENAME;
    &MYFILE = GetFile(&FILENAME, "E", %FilePath_Absolute);
Else
    &MYFILE = GetFile(&FILENAME, "E", %FilePath_Relative);
End-If;
```

optfilename

If you submit a file to Application Engine as the first parameter in the command line, Application Engine reads the contents of the file and interprets the contents as if it were parameters entered on the command line. This option is intended mainly for the Microsoft Windows NT or UNIX Process Scheduler server environment. For example, you might enter *psae \$temp/myparmfile.txt*

Note. For security reasons, after Application Engine interprets the contents of the parameter file, it immediately deletes the file.

Debugging Application Engine Programs

This section discusses how to:

- Enable the Application Engine debugger.
- Set debugging options.

Enabling the Application Engine Debugger

To run a program in debug mode:

1. Set the debug option.

You can set the debug option in the following locations:

- Start PeopleSoft Configuration Manager and select the Process Scheduler tab.

In the Application Engine group, enable debug by selecting the Debug check box. This is the method that applies to all methods of invocation.

- If you used the command line option to invoke your Application Engine program, then you can include the `-DEBUG Y` parameter in the command line you submit to `PSAE.EXE`.

If you already have the Debug check box selected in PeopleSoft Configuration Manager, then you do not need to include the `-DEBUG` parameter in your command line.

Note. Setting debug capabilities in either PeopleSoft Configuration Manager or the command line turns debug mode on. However, if you have debug enabled in Configuration Manager and you submit `-DEBUG N` on the command line, the PeopleSoft Configuration Manager setting defines your default command line value, and the command line can override the default.

- If you have PeopleCode in your Application Engine program, enable the PeopleCode debugger.

When you launch your program and the PeopleCode action executes, you enter the PeopleCode debugger.

2. Execute the Application Engine program to debug.
3. At the Application Engine Debugger prompt, enter a command to enable a debugging option.

Each command is represented by a single letter, such as X, L, or M. Enter the letter that corresponds to the option you want to engage. To see a list of the available debugging options, enter `?` at the prompt.

To enable the PeopleCode debugger for Application Engine:

1. Sign on to PeopleTools using the same user ID that you are going to use to invoke the Application Engine program.
2. Open Application Designer.
3. Select Debug, PeopleCode Debugger Mode.

Your Application Engine program can be open on the desktop, but you do not need to open the Application Engine program or the PeopleCode action that you want to debug.

4. Select Debug, Break at Start.

This causes the Application Engine program to break prior to executing any PeopleCode programs within it.

Setting Debugging Options

Each debugger option is represented by a single letter that you specify at the prompt. To engage the option you select, press Enter.

Debugging Tips

Note the following tips about debugging programs:

- In some cases, such as when setting breakpoints or watch fields, submenus offer more options.

After you are familiar with the commands, you can enter multiple items on the command line to combine commands and bypass the submenus. For example, to see a list of the breakpoints, you could enter `B L`.

To set a field as a watch field, you could enter `W S MY_FIELD`.

Or, if it's on a different state record, enter `W S MY_AET.MY_FIELD`.

Note. The exception to this option is **Modify**, which always displays the current value and then prompts you to enter the new value. You can, however, enter `M MY_AET.MY_FIELD` to get directly to the new value prompt.

- The letter commands are not case-sensitive.

For example, `Q` and `q` are valid commands.

Debugging Options

Option	Description
Quit	<p>Enter <code>Q</code>. This option performs a rollback on the current unit of work in the debugging run, and it ends the debugging session. It effectively terminates your Application Engine program.</p> <p>Quit is useful for testing restart. Have some work committed and some uncommitted. Then, terminate the program at that point and roll back the pending work. You want to make sure the program restarts from the point of the last successful commit.</p>
Exit	<p>This option is valid only after one step has completed and another has not already begun. It is not valid once you reach the action level.</p> <p>Use this option as an alternative to Quit. Exit ends the program run and the debugging session, but it also commits the current unit of that the program has already completed. This option can be helpful when testing your restart logic.</p>
Commit	<p>Enter <code>C</code>. To commit the current unit of work in your program, use this option. It is valid only after a step has completed and before another has already begun. It is not valid once you reach the action level.</p> <p>You can use this option, for example, to use your database query tool to check the data in your database.</p>

Option	Description
Break	<p>Enter B. Sets a breakpoint. When the program reaches the breakpoint, it temporarily halts execution to enable you to observe the state of the current process.</p> <p>Breakpoint options include:</p> <p>Set: Enter S to set a breakpoint location.</p> <p>The breakpoint location defaults to the current location in the program, but you can specify other sections or steps by overriding the default values that appear in the brackets.</p> <p>Unset: Enter U to remove breakpoints previously set.</p> <p>List: Enter L to list breakpoints. When you enter this command, make sure that you have entered B first to specify the break option. If you just enter L from the main command prompt, you engage the Look option.</p>
Look	<p>Enter L. Enables you to observe the values currently in the state record associated with the program you are debugging. You must specify the state record at the Record Name prompt. By default, the default state record as specified in your program properties appears with the brackets.</p> <p>You can also specify a specific field name on the state record in the Field Name prompt. To look at all the fields in the state record, leave the asterisk (*) within the brackets unchanged.</p>
Modify	<p>Enter M. Enables you to modify the value of a state record value for debugging purposes. Suppose the previous steps did not set a value correctly. However, you may want to see how the rest of the program would perform if the appropriate value existed in the state record. This enables you to give your program some help in the debugging or testing phase.</p> <p>As with the Look command, you must specify the appropriate state record (if you are using multiple state records), and you must specify one field. You can modify only one field at time.</p>
Watch	<p>Enter W. When you specify a field as a watch field, the program stops when the value of the field changes.</p> <p>Similar to the Break command, you can specify options for Set, Unset, and List.</p>
Step Over	<p>Enter S. Executes the current step to completion and stop at the next step in the current section.</p> <p>The behavior depends on the current level or the program. You start at the step level, and then can step into the action level. If you are at the step level and use step over, you go to the next step in the current section, skipping over all actions (including any call sections). If you are at the action level, step over executes the current action and stops at the next action in the current step, or at the next step in the current section.</p>

Option	Description
Step Into	<p>Enter I. Use this option to observe a step or called section in a more granular level. For instance, you can check each SQL statement and stop. By using this option and checking the state record at each stop, you can easily isolate problem SQL or PeopleCode.</p> <p>As with Step Over, the behavior depends on the level. At the step level, you can step into the action level and stop before the first action in the step. At the action level, if the current action is a call section, Step Into takes you to the first step in the called section. For other action types, Step Into acts the same as Step Over, because there is no deeper level in which to step.</p>
Step Out of	<p>Enter O. After you've stepped into a step or called section, use the Step Out of option to run the rest of the current step or called section and stop. As with the previous step options, the behavior of Step Out of depends on the current level of the program.</p> <p>At the step level, Step Out of completes the remaining steps in the current section, returns to the calling section or step, and stops at the next action in that step. If the section is MAIN and is not called by another section or step, then Step Out of behaves the same as the Go option.</p> <p>At the action level, Step Out of completes the current step and stops at the next step in the current section, or if the program is at the end of a section, Step Out of returns to the calling section or step.</p>
Go	<p>Enter G. After the program has stopped at a specific location, and you've examined its current state, you can use the Go command to resume the execution of the program. This is a helpful command when you have breakpoints set. With this command, the program won't stop at a step or action; it only stops at the next breakpoint or watch field, or when the program runs to completion.</p>
Run to commit	<p>Enter R. Resumes execution of your program after it has stopped. This command forces the program to stop again after the next commit. This is a good option to use when observing your commit strategy and how it will affect a restart.</p>

Example of the Look Option

To view the value stored in a specific field of the state record after a step or action, enter the appropriate field name at the Field Name prompt. For example, if you entered AE_TESTAPPL_AET at the Record Name prompt and AE_INT_6 at the Field Name prompt, you would see the value of the AE_INT_6 field in the AE_TESTAPPL_AET record.

You can also use an asterisk (*) as a wildcard to get a partial list. For example, if you enter AE_INT* at the Field Name prompt, you see only the fields that start with AE_INT. This is also true for the Record Name prompt. This is useful both to list multiple fields across multiple records or as a shortcut. If you know there is only one state record that starts with XXX, you don't have to type the full name—just enter XXX.

Example of the Modify Option

If you wanted to set the AE_INT_15 field in the AETESTPROG to 10, you would enter the record (AE_TESTAPPL_AET) at the Record Name prompt and the field (AE_INT_15) at the Field Name prompt.

You then see the current value of the field. At the prompt, you can enter a new value.

Using the Look command, you can check to see that the value you specified now exists in the state record.

Example of the Watch Option

Enter S to set a watch field. After you enter S, you enter the record name (such as AE_TESTAPPL_AET) and field name (such as AE_INT_7) at the appropriate prompts.

Enter U to unset, or remove, a watch field from the list. After you enter U, you see a list of active watch fields. You enter the watch field ID number to remove a field. For example, if the field AE_INT_7 were second in the watch field list, you would enter 2 to remove it.

After the completion of a step or action, enter L to list, or view, the values of all the fields that you have included in the watch list.

Note. You cannot set a watch on a long text field.

Restarting Application Engine Programs

One key feature of Application Engine is its built-in checkpoint and restart capabilities. If there is an abnormal termination or failure on a step in the program, you can restart the request from the last successful checkpoint, or the step immediately preceding the step that failed. You restart the program from the process request page.

This section provides an overview of restart and discusses how to:

- Determine when to use restart.
- Control abnormal terminations.
- Restart Application Engine programs.
- Start Application Engine programs from the beginning.
- Enable and disable restart.

Understanding Restart

Application Engine programs save to the database (perform a commit) only when an entire program successfully completes. You must set any individual commits where appropriate.

At the section level, you can set a commit after each step in that section. At the step level, you can require or defer commits for individual steps, or you can increase the commit frequency within a step to *N* iterations of a looping action within a step, such as a Do Select or Do While action.

The commit level that you select plays a major role in how restart works in a program. Each time that Application Engine issues a commit with restart enabled, it records the current state of the program. The recording of the current state that Application Engine performs is referred to as a checkpoint.

Using the restart feature enables you to perform commits more often in a program. Restart reduces the overall impact on other users and processes while the background program is running, because it reduces the amount of rows that are locked by the program, allowing multiple instances of the program to run concurrently (parallel processing), which may be useful for high-volume solutions.

With restart, if a failure occurs at any point in the process, the user can restart the program and expect the program to behave in the following manner:

- Ignore the steps that have already completed up to the last successful commit.
- Begin processing at the next step after the last successful commit.

The ability for Application Engine to remember completed steps depends on a record called AERUNCONTROL, which is keyed by process instance.

When a program runs, each time Application Engine issues a commit it also saves all of the information required for a program restart in the AERUNCONTROL record.

Determining When to Use Restart

Usually, you want to develop programs to take advantage of Application Engine restart capabilities. Programs that are good candidates for restart do a lot of preparation work up front, like joining tables and loading data into temporary work tables. Also, programs that might put data in an unstable state if they terminate abnormally during a run should be considered to take advantage of restart. As a general rule, restart is essential for programs that primarily do set-based processing.

However, if your program has one the following characteristics, you may want to disable restart:

- It is mainly row-by-row processing.
- The overhead involved with Application Engine performing a checkpoint during the program run is not desirable.
- The program commits after N iterations of a looping construct within a step, and the Select statement driving the loop is composed in such a way that if the program terminated and then started again, it would ignore transactions that were already processed in the previous program run. In this sense, the program processes the restart internally, in that Application Engine treats each start of a program as a fresh start, instead of restarting a previous instance.

When developing for restart, consider the consequences if a program fails and you can't restart the program. Given the commit structure that you've defined for your Application Engine program, would your data remain in an usual state if a failure were to occur after any of the commits? Would it be easy to recover from such a case?

Using Restart at the Program Level

Application Engine automatically performs all state record updates. When an Application Engine program starts, it inserts a row in the state record for the assigned process instance. Then the system updates the state record whenever the program performs a commit to store changed values into the database. Finally, the state record row is deleted upon successful completion of the application.

However, if the state record the program uses is a work record, no database updates can be made to the record. Consequently, if you restart the program, you might get unexpected results, because the memory was lost when the program terminated. In fact, the system reinitializes any state records that are work records at each commit, to ensure consistent behavior during a normal run and a restarted run. Therefore, you may need to make at least one of your state records a SQL table to contain values that must be retained across commits or in case of termination.

Finally, the other consideration for programming for restart at the program level is to check both the Application Engine Program Properties dialog box and PeopleSoft Configuration Manager to make sure that Disable Restart check box is not selected.

Using Restart at the Section Level

The section level property associated with restart is section type, which has the options Prepare Only and Critical Updates.

If a section is only preparing data, as in selecting it, populating temporary tables, or updating temporary tables, then set the section type to Prepare Only. However, if the section updates permanent application tables in the database, set the option to Critical Updates.

During runtime, when the system arrives at the first section set to Critical Updates, it sets the AE_CRITICAL_PHASE value in the AERUNCONTROL record to *Y*. Once set, the value of AE_CRITICAL_PHASE remains *Y* until the program completes successfully. When the program completes, the corresponding row in AERUNCONTROL is deleted. Therefore, a Prepare Only section following the Critical Updates section won't reset the AE_CRITICAL_PHASE value to *N*.

If your program terminates, the user can check the AE_CRITICAL_PHASE value. If it's *Y*, the user knows that the section that failed is critical and that the program should be restarted to ensure data integrity. If AE_CRITICAL_PHASE is *N*, restarting may not be necessary; however, as a general rule, you should restart even if AE_CRITICAL_PHASE is set to *N*.

Using Restart at the Step Level

In your program's Where clause of a Do Select action, you should include conditions that reduce the answer set returned from the Select statement.

For example,

```
SELECT RECNAME, FIELDNAME
  FROM PS_AE_RECFIELD
 ORDER BY RECNAME, FIELDNAME
```

If you ran this Select statement as part of a Do Select action with *Restartable* selected as the Do Select type, the system might process some of the rows twice after a restart. Also, if you have specified *Reselect*, the program could execute in an infinite loop, because there's nothing to reduce the answer set. However, if you modified the Select statement to look more like the following, you could make it *Restartable*.

```
SELECT RECNAME, FIELDNAME
  FROM PS_AE_RECFIELD
 WHERE RECNAME > %Bind(RECNAME)
 OR (RECNAME = %Bind(RECNAME) AND FIELDNAME > %Bind(FIELDNAME))
 ORDER BY RECNAME, FIELDNAME
```

A Do Select action that has been coded for *Restartable* can be converted to *Select/Fetch*, but the opposite is not true.

The previous example shows the use of a key column to reduce the answer set. This can be convenient if your record has only one or two key fields. However, if your record has three or four keys, your SQL would become overly complex.

Instead of matching key fields, you could add a switch to the selected table, and then have the processing of the called section modify the switch as it processes the row. In this example, your Select statement could look like the following:

```
SELECT COLUMN1, COLUMN2, . . .  
FROM PS_TABLE1  
WHERE PROCESSING_SWITCH='N' . . .
```

Controlling Abnormal Terminations

A controlled abnormal termination (sometimes called an *abend*) means that Application Engine exits gracefully because of a calculated error condition. Some examples of controlled abends are:

- SQL errors while you have set On Error to *Abort*.
- A PeopleCode return value, when On Return is set to *Abort*.
- A SQL statement that affects no rows, when you have set On No Rows to *Abort*.

In these situations (when Application Engine is in control) the Run Status field in Process Monitor reads *Error*.

An uncontrolled termination occurs when there is a memory violation or a user terminates a process. In these cases, the Run Status field in Process Monitor shows *Processing*.

Restarting Application Engine Programs

You can restart an Application Engine program in one of these ways:

- From the command line.
- From a process request page.

Note. The following procedures for restarting a failed Application Engine program assume that you have rectified the error that caused the program to fail in the first place. For instance, suppose the name of a referenced table has changed. Regardless of how many times you restart the program, it will continue to fail until you've modified references to the old table name.

Restarting from the Command Line

Normally, only developers and system administrators use the command line for restarting Application Engine programs. Users, in most cases, should not be expected to use this method.

You can use the command line option to restart programs that run on the client or the server. Application Engine references only the process instance of the failed process. Therefore, if you run a process on the client and it fails, you can restart it from the server using the server command line. Likewise, if you run a process from the server and it fails, you could restart it from the client using the command line.

To restart an Application Engine program from the command line:

1. Collect the command line values associated with the failed program.

These values include database type, database name, user ID and password, run control ID, program name, and the process instance. You can find these variables on the Process Details dialog box, the corresponding state record, or the Application Engine Run Control table. Where the values reside depends on how you invoked the program. For instance, if you invoked the program using the command line, or outside of PeopleSoft Process Scheduler, then you cannot view details associated with the program run on the Process Details dialog box.

2. Enter the following command line syntax at the command prompt substituting the values from the previous step:

```
PSAE.EXE -CT DB_TYPE -CD DB_NAME -CO OPRID -CP PASSWORD -R
RUN_CONTROL -AI PROGRAM_NAME -I PROCESS_INSTANCE
```

Note. Some database platforms, such as Sybase, also require that you include a server name in the argument list.

Restarting from the Process Request Page

You can restart programs from a process request page only for those programs that run on the server.

To restart an Application Engine program from a process requests page:

1. Open PeopleSoft Process Scheduler by selecting PeopleTools, Process Scheduler, System Process Requests.
2. Locate the run control ID number of the program to restart.
3. To display the details of the failed process, click the Process Detail link.
4. On the Process Request Details page, select Restart Request, and click OK.

Bad Restart Error

If you attempt to restart what Application Engine believes to be a process that completed successfully, you receive a bad restart message. You can also get this message if your Application Engine application is defined with restart disabled.

Starting Application Engine Programs from the Beginning

When an Application Engine program ends abnormally, you may have to decide whether you should restart the process or just start it from the beginning. As your Application Engine program ran at least part way through, starting over may leave your data in an unknown state. Also, application logic might need to be undone, depending on what stage the program was at when it failed, what data the program had committed, and so on.

However, if restart is enabled and you attempt to start a new process that matches the run control ID and user ID for another process, you receive a suspend error. Because the process instance for these two processes is different, the new request fails. This usually occurs when a user tries to run the program again after receiving an error on the previous attempt.

To start the program over from the beginning, you can use SQL to delete the row that corresponds to the failed program from the Application Engine run control table and your state record.

To restart an Application Engine program from the beginning:

1. Open your native SQL editor and manually delete the row in PS_AERUNCONTROL that corresponds to the program you want to start from the beginning.

Use the following SQL to accomplish this step:

```
DELETE FROM PS_AERUNCONTROL
WHERE OPRID=OPRID
AND RUN_CNTL_ID=Run_Control_ID
```

2. Delete from your state record the row that corresponds to the failed program run.

Use the following SQL to accomplish this step:

```
DELETE FROM PS_MY_AET
WHERE PROCESS_INSTANCE=Process_Instance
```

Note. To restart the program, you can also select Restart Request from the Process Request Details dialog box.

Enabling and Disabling Restart

To disable restart, use any of these methods:

- Select the Disable Restart check box on the Application Engine Program Properties dialog box.

To access program properties, select File, Definition properties, and select the Advanced tab.

- Select the Disable Restart check box in the Configuration Manager profile.

To access the profile, start Configuration Manager, select the Profile tab, and click Edit. Then select the Process Scheduler tab.

- Include the -DR Y option in the command line of PSAE.EXE.

If you disabled restart in any of these three places, restart is disabled.

Therefore, if you want the program to restart in a production environment while still keeping a restart condition from getting in the way during development and testing, you can deselect the Disable Restart check box in the Application Engine program properties. Then, during development, you can select the Disable Restart check box in Configuration Manager or invoke your program from the command line with the -DR Y option, without having to reconfigure the program for testing.

Caching the Application Engine Server

Application Engine caches metadata, just like the application server. This caching enhances performance, because a program can refer to the local cache for any objects that it uses.

Cache Directory Location

Application Engine programs that run on a Microsoft Windows NT or UNIX server each lock their own cache directory for the duration of the run. These directories are found under the master cache directory. The master directory is created under the directory specified by the CacheBaseDir variable in the PeopleSoft Process Scheduler configuration file. If all existing cache directories are locked, a new one is created. Cache subdirectories are named sequentially, starting at 1.

If you do not enter a fully qualified path for the CacheBaseDir variable, then Application Engine creates the cache directory within the directory in which the program is set to run.

Note. Do not share the CacheBaseDir variable with application servers, and do not use environment variables when specifying CacheBaseDir, because the system does not resolve them. For example, do not set CacheBaseDir to \$PS_CFG_HOME.

Cache Parameters

The PSPRCS.CFG (PS_SERVER_CFG) file has two additional cache parameters. They are:

- Enable Server Caching
- Server Cache Mode

Do not alter these settings from the delivered defaults. These settings are reserved for future use.

Freeing Locked Temporary Tables

If you use dedicated temporary tables for Application Engine programs, then you might need to free, or unlock, a temporary table if the program running against it terminates abnormally. Because most Application Engine programs run through PeopleSoft Process Scheduler, typically you use Process Monitor to unlock the temporary tables. Deleting or restarting a process using Process Monitor automatically frees the locked temporary tables.

For the programs that you invoke outside of PeopleSoft Process Scheduler, use the Manage Abends page. Programs running outside of Process Scheduler include those invoked from CallAppEngine PeopleCode and the command line.

To free locked temporary tables using the Manage Abends page:

1. Select PeopleTools, Application Engine, Manage Abends.
2. Identify the program that has the particular temporary tables locked.

You can uniquely identify programs using the process instance, run control ID, program name, user ID, and run date and time columns.

3. Click the Temp Tables link.
4. On the Temporary Tables page, click the Release button to unlock the temporary tables associated with the program.

Chapter 7

Calling Application Engine Programs from COBOL

To facilitate the conversion of existing COBOL programs to Application Engine programs, you can call Application Engine programs from existing COBOL code.

This chapter discusses how to:

- Add copybooks to COBOL programs.
- Assign copybook values.
- Handle COBOL errors.

Adding Copybooks to COBOL Programs

To enable you to call Application Engine programs from COBOL programs, include the copybook called PTCCBLAE.CBL with your COBOL programs. This copybook is located in *PS_HOME*\src\cbl\base.

The following is the PTCCBLAE.CBL copybook.

```

*01  CBLAE.

NOCLN      02  CBLAE-PRCSNAME          PIC X(12)    VALUE SPACE.

NOCLN      02  CBLAE-COMMIT-FLAG        PIC X(1)     VALUE SPACE.
           88  AE-COMMITS-SUCCESS      VALUE 'B'.
           88  AE-COMMITS-ALL           VALUE 'C'.

           02  CBLAE-PARMS.
               03  CBLAE-PARM-CNT        PIC 9(4)          COMP.
               03  CBLAE-PARM-ENT        OCCURS 500 TIMES.
                   05  CBLAE-STATEREC    PIC X(15).
                   05  CBLAE-FIELDNM     PIC X(18).
                   05  CBLAE-DATA-PTR     POINTER.
                   05  CBLAE-LENGTH      PIC 9999          COMP.
                   05  CBLAE-SCALE       PIC 99            COMP.
NOCLN      05  CBLAE-TYPE              PIC X.
           88  CBLAE-TYPE-CHAR          VALUE 'C'.
           88  CBLAE-TYPE-SMALLINT      VALUE 'S'.
           88  CBLAE-TYPE-INT           VALUE 'I'.
           88  CBLAE-TYPE-DEC           VALUE 'P'.
           88  CBLAE-TYPE-DATE          VALUE 'D'.
           88  CBLAE-TYPE-TIME          VALUE 'T'.
           88  CBLAE-TYPE-TIMEONLY      VALUE 'V'.
           88  CBLAE-TYPE-NUMERIC       VALUE 'S' 'I' 'P'.

```

Data Transfer Process Between COBOL Programs and Application Engine Programs

To interface between COBOL programs and Application Engine programs, the process uses a file to pass parameters from COBOL to the Application Engine program. This file is owned by the process and has the *prm* extension. The location of the file is determined by the following:

- If an application server root directory is defined, the file resides in the output directory of that particular process instance.
- If the output directory on the application server is not defined, the file resides in the default output directory of the Process Scheduler domain..
- If neither one of the above is defined, the file is written to the default temp directory.

Assigning Copybook Values

To assign values to the copybook of the calling COBOL program, to be passed as parameters into the state records of the called Application Engine program:

- Identify the fields in your COBOL program that contain the values you want to pass to the Application Engine program.
- Load the PTCCBLAE.CBL copybook with the state record name, field name, field length (this should be the size of the field not the size of the contents), the scale (decimal places if any), and set the field type.
- Call the PTPSETAD program to set the pointer in PTCCBLAE.CBL to the host programs variable.
- Set the variable AE-COMMIT-FLAG to either AE-COMMITS-ALL or AE-COMMITS-SUCCESS.

AE-COMMITS-ALL means that the Application Engine program commits as specified in the program.
AE-COMMITS-SUCCESS means that the Application Engine program ignores all commits and performs one commit at the end of successful execution.

Example of Loading Values from PTPSTAE.CBL Sample Program

Make sure the calling COBOL program has successfully connected to the database before calling the PTPCBLAE copybook, and ensure that the calling program is not running through a RemoteCall function.

The following code shows an example of how to load values from the copybook:

```

MOVE 0 TO CBLAE-PARM-CNT OF CBLAE

      ADD 1 TO CBLAE-PARM-CNT OF CBLAE

MOVE 'QE_CBLAETST_AET' TO CBLAE-STATEREC
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

MOVE 'DESCR' TO CBLAE-FIELDNM
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

MOVE 30 TO CBLAE-LENGTH
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

MOVE 0 TO CBLAE-SCALE
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

SET CBLAE-TYPE-CHAR OF CBLAE (CBLAE-PARM-CNT OF CBLAE)
      TO TRUE

CALL 'PTPSETAD' USING CBLAE-DATA-PTR
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)
      W-DESCR OF W-WORK

      ADD 1 TO CBLAE-PARM-CNT OF CBLAE

MOVE 'QE_CBLAETST_AET' TO CBLAE-STATEREC
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

MOVE 'QE_AE_INT_7' TO CBLAE-FIELDNM
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

MOVE 2 TO CBLAE-LENGTH
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

MOVE 0 TO CBLAE-SCALE
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)

SET CBLAE-TYPE-SMALLINT
OF CBLAE (CBLAE-PARM-CNT OF CBLAE)
      TO TRUE

CALL 'PTPSETAD' USING CBLAE-DATA-PTR
      OF CBLAE (CBLAE-PARM-CNT OF CBLAE)
      W-SMINT OF W-WORK

```

*

```
DA000-CALL-AE SECTION.  
  
DA000.  
  
*  
  
MOVE 'QE_AETESTPRG' TO CBLAE-PRCSNAME OF CBLAE  
  
SET AE-COMMITS-ALL TO TRUE  
  
CALL 'PTPCBLAE' USING SQLRT CBLAE.  
  
CALL-AE-EXIT.  
  
EXIT.
```

Sample of the Communication Area of PTPBLAE.CBL

If the called Application Engine program updated the state records or fields that were passed by PTPCBLAE, the fields or records are stored in the local variables of the calling program as identified by PTPSETAD.

```

* PTCCBLAE - Communication area for PTPCBLAE
*
*01  CBLAE.
NOCLN  02  CBLAE-PRCSNAME  PIC X(12)  VALUE SPACE.
*      Name of AE program to be called.
NOCLN  02  CBLAE-COMMIT-FLAG  PIC X(1)  VALUE SPACE.
*      Flag to determine which of the following commits to make.
        88 AE-COMMITS-SUCCESS          VALUE 'B'.
*      No in-process commit; if successful, then commit occurs.
        88 AE-COMMITS-ALL              VALUE 'C'.
*      Commits occur when defined in the AE program.
        02  CBLAE-PARMS.
            03  CBLAE-PARM-CNT  PIC 9(4)COMP.
*      Counter of the number of state records passed.
            03  CBLAE-PARM-ENT  OCCURS 500 TIMES.
*      Maximum value of state record entries.
            05 CBLAE-STATEREC  PIC X(15).
*      State record name.
            05 CBLAE-FIELDNM  PIC X(18).
*      Field name.
            05 CBLAE-DATA-PTR  POINTER.
*      Pointer to your own working storage area.
            05 CBLAE-LENGTH  PIC 9999  COMP.
*      Field length of defined state record.
            05 CBLAE-SCALE  PIC 99  COMP.
*      Number of decimal places.
NOCLN      05 CBLAE-TYPE          PIC X.
*      Field data type.
            88  CBLAE-TYPE-CHAR          VALUE 'C'.
            88  CBLAE-TYPE-SMALLINT      VALUE 'S'.
            88  CBLAE-TYPE-INT           VALUE 'I'.
            88  CBLAE-TYPE-DEC           VALUE 'P'.
            88  CBLAE-TYPE-DATE          VALUE 'D'.

```

```
88  CBLAE-TYPE-TIME          VALUE 'T' .  
88  CBLAE-TYPE-TIMEONLY     VALUE 'V' .  
88  CBLAE-TYPE-NUMERIC      VALUE 'S' 'I' 'P' .
```

Handling COBOL Errors

If your COBOL program needs error handling, try the following procedure:

1. Add a field (return code) to your state record.
2. Initialize the field to a negative value.
3. Pass the value into the Application Engine program.
4. At the successful completion of the Application Engine program, change the field value to a positive value.
5. Check for that value in your COBOL program.

Chapter 8

Tracing Application Engine Programs

This chapter provides overviews of tracing Application Engine programs and trace results and discusses how to:

- Enable Application Engine tracing.
- Locate trace files.

Understanding Tracing Application Engine Programs

You can set the following traces to monitor the performance of Application Engine programs:

- Application Engine step trace.
- Application Engine SQL trace.
- Application Engine statement timings trace.
- Database optimizer trace.

Note. The general PeopleTools SQL and PeopleCode traces also apply to Application Engine programs.

Understanding Trace Results

This section discusses:

- Trace file sections.
- Step trace.
- SQL trace.
- Statement timings trace.
- Database optimizer trace.

Trace File Sections

At the top of each trace, useful information helps you to identify the PeopleTools version, the database name, and the database platform type.

SQL Counts and Timings Section

The first section of a trace file is the SQL section. It records the performance of application-specific SQL. The trace values appear within a series of columns and sections. The following table describes each column within the first section of the trace file:

Column	Description
SQL Statement	Application Engine SQL actions and stored SQL objects always have a statement ID. The SQL Statement column shows the statement ID, so that you can attribute trace values to individual SQL statements. In the case of SQLExec SQL, a portion of the SQL statement appears in the first column, to help you identify it. For SQL objects, use the TraceName property in the Create SQL so that you can uniquely identify it in the traces.
Compile Column	This column shows how many times the system compiled a SQL statement and how long it took. The term <i>compiled</i> refers to the SQL statement being sent to the database to be parsed and optimized, and it also includes the time required for the first resolution of any PeopleSoft meta-SQL.
Execute Column	This column shows how many times the system executed the SQL statement and the time consumed doing so. The term <i>executed</i> refers to the system sending the compiled SQL to the database server to be run against the database.
Fetch Column	This column applies to Select statements. It shows how many rows your program fetched from the database and how much time this consumed. The system must first execute a Select statement against the database to find the relevant rows and generate an active set. After the set exists, the program must still fetch the rows. Some database APIs have buffered fetches, which means that the fetch may include more than one row. Therefore, subsequent fetches are free until the buffer becomes empty.
Total Column	This column shows the sum of the compile, execute, and fetch times of the SQL statement. Some database APIs may defer a compile to the execute phase, or defer an execute to the first fetch operation.
PeopleCode SQL	This subsection is for SQL executed from PeopleCode actions. Compile counts and times for such SQL is included in execute count and times, because you do not explicitly control the ReUse feature. To determine whether ReUse is occurring, you must do a program run after enabling the generic PeopleTools trace for SQL statements, API calls, and so on. As a starting point, use a trace value of 31.

Column	Description
Application Engine SQL	<p>This subsection reveals the time attributed to Application Engine overhead that is not directly related to the SQL within your program. For example, the values in this section represent the SQL generated for checkpoints, commits, and so on. If there are Commit statements without checkpoints, it indicates that restart has been disabled, or a restartable program has called a non-restartable program.</p> <p>If the time consumed performing a checkpoint or committing seems more than expected, you should try to reduce it if possible by setting the commit frequency of the steps containing Do loops.</p>
AE Program: <i>program_name</i>	<p>This subsection shows SQL actions for a particular program. The action properties that affect performance are flagged. For example, BulkInsert. ReUse is not flagged because it is self-evident when the Execute count is higher than the compile count.</p>

Note. When you run a SQL trace at the Application Engine level and the PeopleTools level simultaneously, you may see misleading results. Extra overhead is added to the overall SQL timings by the PeopleTools trace. Tracing SQL at the Application Engine level (-TRACE) adds to the non-SQL times because PeopleTools writes the trace data after timing the SQL.

PeopleCode Actions Section

The second section of the trace file, or PeopleCode section, records the performance associated with all the PeopleCode actions in your program. The following table describes each column in this section:

Column	Description
PeopleCode	This column contains the names of the PeopleCode actions in your program.
Call	This column shows how many times each PeopleCode action is called during the program run.
Non-SQL	This column shows the time spent by your PeopleCode actions that does not involve any SQL.
SQL	This column shows the time spent by your PeopleCode actions executing SQL. The SQL time total should be similar to that of the PeopleCode SQL subsection in the first section of the trace file.
Total	The total indicates the cumulative amount of time spent in the action.

Note. The system rounds to the first decimal place (tenths), but only after calculating the sum of each action time.

PeopleCode Built-ins and Methods Section

The third section of the trace file contains either a list or summary of PeopleCode built-ins and methods used. To see a list of built-ins and methods, you must enable the PeopleCode detail timings in addition to the statement timings trace.

If a method or built-in function takes a large amount of time, you may want to consider alternatives. For example, if array processing dominates your runtime, consider inserting the data into temporary tables and performing the processing on tables in the database.

Summary Data

The fourth section of the trace file contains summary data. The values in this section reveal an overview of the program run without drilling down too far into details.

The following table describes the values that appear in this section:

<i>Column</i>	<i>Description</i>
Total run time	This value presents the overall amount of time a program required to complete from start to finish.
Time in application SQL	This value represents the time that your program spent executing SQL. The value includes SQL executed by both PeopleCode and SQL actions.
Percent time in application SQL	This value represents the percentage of time spent executing SQL compared to the entire program run.
Time in PeopleCode	This value represents the time that your program spent executing PeopleCode. Time in PeopleCode excludes SQL executed from within PeopleCode.
Percent time in PeopleCode	This value represents the percentage of time spent executing PeopleCode compared to the entire program run.
Total time in Cache	This value represents the amount of time your program spent retrieving objects from the cache or refreshing the cache. Time in cache includes all memory cache access, file cache access, and SQL executed to load managed objects, such as Application Engine program components, metadata, and so on. Time varies according to where Application Engine finds an object. For instance, retrieving an object that the system cached during a previous run is faster than retrieving it from the database.
Number of calls to Cache	This value represents the actual number of calls your program made to the cache. The number of calls to the cache remains constant for the same Application Engine program processing the same data.

Environment Information Section

The fifth section of the trace file contains environment information specific to Application Engine. If programs appear to be performing poorly, check the trace value that you have set.

Each trace produces an unavoidable degree of overhead. As such, the more traces you have enabled, the more likely you are to see degraded performance. Run only the traces you need. This section of the trace file shows you information about the following:

- SQL trace
- PeopleCode trace
- Application Engine trace
- Application Engine DbFlags (%UpdateStats)

Step Trace

The step trace reports each step name that your program executes and in what order. Associated with each step is a timestamp, the Do action level, and the action type.

The trace shows the steps that execute within a called section by indented formatting. For example, a step that executes within a called section is preceded by two dots (..), while other steps are preceded by only one dot.

SQL Trace

The SQL trace report shows formatted SQL processes, including commits, rollbacks, and restarts. You can also view the buffers associated with each SQL statement. Use the SQL trace to spot errors in your SQL and to view your commit strategy.

Statement Timings Trace

The Application Engine statement timing trace report is similar to a COBOL timings trace, in which you monitor the execution of COBOL programs for performance evaluations. This trace enables you to gather performance information to determine program bottlenecks. Once bottlenecks are identified, you might be able to modify your program to run more efficiently, or you may want to change the database schema and configuration to optimize the execution of your program.

The statement timings trace is invaluable for tuning an Application Engine program. It may also be useful as a default trace level for all production runs, to provide a metric for long-term performance trends.

By examining all of the figures in this trace, you can identify areas of your program that are not running as efficiently as possible. For instance, if compile counts are high, you can reduce the numbers by using the Application Engine reuse feature. If inserts appear to be running slow and you have many of them, you can increase the performance by using the Application Engine bulk insert feature. Each value in the trace, including cumulative totals, appears in a form rounded to the nearest tenth of a second, but totals are calculated using nonrounded timings.

You can write this trace to a file, or you can write the results to tables. Either way, timings trace overhead is minimal. Internal testing reveals that the Application Engine trace has an overhead between 2 percent and 5 percent of total runtime.

By storing timings information in a table, you can store historical data in the database, which enables you to produce reports that aid in trend analysis, allow ad hoc SQL queries for longest running statements, and so on. With the timings data stored in the database, you can manipulate and customize reports to show only the metrics in which you are most interested.

You can use third-party tools to query and present the data in such ways as detailed graphical representations of your program's performance. You can also implement alarms if the performance of a program reaches a specified maximum value in a particular area such as SQL compile time.

Note. Application Engine does not write the timings trace to a table for programs invoked by the CallAppEngine PeopleCode function. To write to a table, a process instance is required, and programs invoked by CallAppEngine are not assigned a process instance.

The Statements Timings (table) option, or 1024 -TRACE option, populates the following tables.

- PS_BAT_TIMINGS_LOG (Parent)

This table stores general information for a program run.

- PS_BAT_TIMINGS_DTL (Child)

This table stores details associated with a program run, such as the execute count, fetch time, and so on.

The SQL Timings (table) option, or the 16834 -TRACE option, populates the following tables.

- PS_AE_TIMINGS_LG (Parent)

This table stores general information for a program run.

- PS_AE_TIMINGS_DT (Child)

This table stores details associated with a program run, such as the execute count, fetch time, and so on.

PS_BAT_TIMINGS_FN

This table stores PeopleCode detailed timing information.

PeopleSoft provides BATTIMES.SQR as an example of the type of reports you can generate to reflect the information stored in the BAT_TIMINGS tables. You can produce a summary report for all the programs for a specific run control ID, or you can get detail data for a specific process instance.

To invoke the BATTIMES.SQR report through PeopleSoft Process Scheduler:

1. Select PeopleTools, Process Scheduler, Batch Timings.

The Batch Timings page appears.

2. From the Report Type drop-down list box, select *Detail* or *Summary*.

3. In the Batch Timings For group box, enter the run control ID for summary reports, and enter the process instance for detail reports.

4. When you have made the appropriate selections, click the Run button.

To view batch timings using Process Monitor:

1. Select PeopleTools, Process Scheduler, Process Monitor.
2. Locate the program run associated with the current trace.
3. Click the Job Details button.
4. On the Process Detail dialog box, click the Batch Timings link.

PeopleCode detail timings do not appear. They appear only in the file format.

Database Optimizer Trace

The database optimizer trace reveals the execution or query plan for the SQL that your Application Engine program generates. Each SQL statement is traced only once. You can write the trace to a file or a table.

How you view the results of this trace depends on the relational database management system (RDBMS) that you are currently using. For instance, on some platforms, only the trace-to-file option is available, whereas on others only the trace-to-table option is available. The following table shows the options available for each of the platforms PeopleSoft supports:

<i>RDBMS</i>	<i>Output</i>
Oracle	File and table
DB2 for OS/390	Table
DB2 for UDB (AIX, Sun Solaris, Microsoft Windows NT)	Table
Microsoft SQL Server	File and table
Informix	File
Sybase	N/A

Note. PeopleTools does not collect optimizer data for SQL originating from PeopleCode actions, except if you run Oracle and Informix and use file output. In this case, the system traces all SQL that executes after the first SQL action executes.

Oracle

When outputting the trace to a file, Application Engine writes the trace file to the default Oracle trace directory specified on the database server. To read the trace file, use the TKPROF utility.

To output the trace to a table on Oracle, a `PLAN_TABLE` table must exist, and the `statement_id` must be of type `VarChar2(254)`, instead of `VarChar2(30)`.

When outputting to a table, the PeopleSoft application updates the trace rows as follows:

- **EXPLAIN PLAN SET STATEMENT_ID:** PeopleSoft updates the `STATEMENT ID` column:

```
EXPLAIN PLAN SET STATEMENT_ID = ApplId.Section.Step.Type FOR sqlstmt
```

- **PLAN_TABLE's REMARKS column:** PeopleSoft updates the `REMARKS` column:

```
PLAN_TABLE's REMARKS column = 'ProcessInstance-RunControlId(QueryNo)'
```

queryno is a count of how many SQL statements have been traced up to a particular point.

Note. When tracing to a table with Oracle, PeopleSoft does not perform optimizer traces on `%UpdateStats` and `%TruncateTable` unless the latter resolves into a `Delete` statement. Alternatively, outputting the Oracle `TKPROF` utility to file handles both the `Analyze` and `Truncate` commands.

Microsoft SQL Server

When you output the trace to a file, Application Engine writes the optimizer trace to the following location: `%TEMP%\psms<queueid><spid>.trc`. To read the trace, you use the SQL Server Profiler utility.

Note. The trace file is written to the server directory when you've specified the trace on the client. If the client has `%Temp%` set to a drive or directory that does not exist on the server, Application Engine does not generate a trace file.

When you output to a table, Application Engine writes the trace data to the `dbo.PS_OPTIMIZER_TRC` table. PeopleTools creates the table automatically when you run the trace for the first time. The trace data written to the table is identical to the data appearing in the optimizer trace file.

You also need to use the SQL Server Profiler utility to view the optimizer results. To view the populated trace table, specify the current server and database on the Source Table dialog box. The Owner value must be *dbo*, and the Table value *PS_OPTIMIZER_TRC*.

In the trace, you find information regarding text, duration, and start time for the following:

- Execution plans.
- Remote procedure calls.
- Insert statements (Update, Delete, and Select statements).
- PeopleSoft-generated user events associating trace data with a PeopleSoft SQL identifier.

If the Application Engine program terminates while you are using this trace option, check that Application Engine was not tracing a SQL statement at the moment the program terminated. If it was tracing a SQL statement at that time, you must manually kill the trace. Otherwise the trace thread on the server continues to run and lock the trace file, and each time that server process ID (SPID) is reused by the server, new information is appended to the locked trace file.

To stop the trace manually, submit the following command from Query Analyzer:

```
xp_trace_destroyqueue queueid
```

The *queueid* in the file name %TEMP%\psms_queueid_spid.trc is the ID corresponding to the queue used for the first SQL statement that the system profiled. Because this trace is only designed to trace Application Engine SQL (not PeopleTools SQL), the queue is closed after every statement profiled. Therefore, the queue that must be destroyed may not be the queue ID used in the trace file.

Note. If the %TEMP% variable is set to a location that does not exist, Application Engine does not generate a trace file.

Informix

For Informix, you can only output the trace to a file. The trace file location depends on the operating system on which your database server runs.

- UNIX.

For UNIX, Application Engine writes the plan to the sqexplain.out file. If the client program runs on the same machine as the database server, the sqexplain.out file appears in the current directory. When the current database is on another computer, the sqexplain.out file is written to the PeopleSoft owner's directory on the remote host.

- Microsoft Windows NT.

For Microsoft Windows NT, Application Engine writes the plan to the following file:
INFORMIXDIR%\sqexpln\username.out.

DB2 for OS/390

For DB2 for OS/390, you can only output the optimizer trace to a table. PeopleSoft has implemented the following to facilitate this trace:

- PeopleSoft selects the maximum query number from the PLAN_TABLE table, increments it by 1000 to avoid clashing with other processes, and then increments it by 1 for every SQL statement traced.
- PeopleSoft sets the SET REMARKS parameter to the following value: ApplId.Section.Step.Type-RunControlId(ProcessInstance)

Note. Before using the Database Optimizer Trace, you must first create a DB2 PLAN_TABLE. Refer to your DB2 UDB for OS/390 and z/OS Administration Guide for the proper format and instructions on creating the PLAN_TABLE.

DB2 for UNIX

For DB2 for UNIX, you can only output the optimizer trace to a table. To facilitate this trace for DB2/UNIX, PeopleSoft has implemented:

EXPLAIN ALL SET QUERYNO =ProcessInstance SET QUERYTAG = 'Section.Step' FOR *sql stmt*

Note. Before using the Database Optimizer Trace, you must first create the DB2 explain tables.

Database Optimizer Trace and Performance

While the database optimizer trace is enabled, performance may be affected. Typically, you turn on this trace only when you are collecting detailed performance metrics. When you are not tuning your performance, turn off the optimizer trace.

To prevent an administrator, or perhaps a user, from unwittingly turning the optimizer trace on or leaving it on after doing performance tuning, you can disable the database optimizer trace for an entire database.

For example, suppose you have a production and a development database, you might want to enable the optimizer trace for the development database while disabling the optimizer trace for the production database.

On the PeopleTools Options page, clear the Allow DB Optimizer Trace option to disable the optimizer trace for the database.

Enabling Application Engine Tracing

By default, all Application Engine traces are turned off. To see a trace or a combination of traces, set trace options before you execute the program.

This section discusses how to:

- Set command line options.
- Set parameters in server configuration files.
- Set options in Configuration Manager.

Setting Command Line Options

The command line option is available for Microsoft Windows NT and UNIX, but it is not available when calling Application Engine programs from PeopleCode.

To enable tracing from the command line, specify the `-TRACE` option within the command line that you submit to `PSAE.EXE`. For example:

```
n:\pt840\bin\client\winx86\psae.exe -CT MICROSOFT -CD PT800GES -CO PTDMO?
  -CP PTDMO -R PT8GES -AI AETESTPROG -I 45 -TRACE 2
```

The following table describes the available `TRACE` option parameter values:

<i>Value</i>	<i>Description</i>
0	Disables tracing.
1	Initiates the Application Engine step trace.
2	Initiates the Application Engine SQL trace.

Value	Description
4	Initiates the trace for dedicated temporary table allocation to an Application Engine trace (AET) file. You can trace how the system allocates, locks, and releases temporary tables during program runs.
128	Initiates the statement timings trace to a file, which is similar to the COBOL timings trace to a file.
256	Initiates the PeopleCode detail to the file for the timings trace.
1024	Initiates the statement timings trace and stores the results in the following tables: PS_BAT_TIMINGS_LOG and PS_BAT_TIMINGS_DTL.
2048	Requests a database optimizer trace file.
4096	Requests a database optimizer to be inserted in the Explain Plan table of the current database.
8192	Sets a trace for PeopleSoft Integration Broker transform programs.
16384	Initiates a SQL timings trace and stores the results in the following tables: PS_AE_TIMINGS_LG and PS_AE_TIMINGS_DT.

To specify traces on the command line, you enter the sum of the desired trace options. This is similar to adding the trace values using PSADMIN, such as the COBOL statement timings or the SQL statement trace value. To specify a combination of traces, enter the sum of the corresponding trace values. For example, to enable the step (1), the SQL (2), and the statement timings (128) traces, you would enter 131—the sum of 1, 2, and 128.

To disable tracing, explicitly specify `-TRACE 0`. If you don't include the `-TRACE` flag in the command line, Application Engine uses the value specified in the Process Scheduler configuration file or in Configuration Manager. Otherwise, the command-line parameters override any trace settings that may be set in Configuration Manager.

Setting Parameters in Server Configuration Files

You can also enable traces in the configuration files for both the application server and the PeopleSoft Process Scheduler server.

For programs invoked by PeopleCode and run on the application server, set the `TraceAE` parameter in the Trace section of the Application Server configuration file (PSAPPSRV.CFG). You can use PSADMIN to set this parameter.

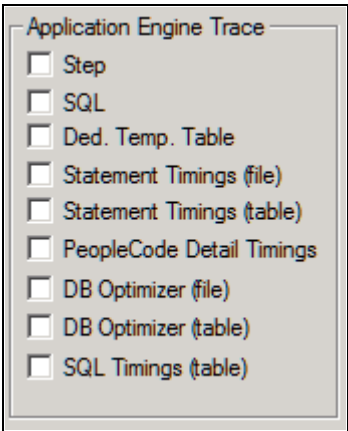
In the PeopleSoft Process Scheduler configuration file, set the `TraceAE` parameter in the Trace section to indicate a level of tracing. You can use PSADMIN to set this parameter.

This option is available on Microsoft Windows NT and UNIX, and applies only to Application Engine programs invoked in batch mode.

Note. The TraceFile parameter does not specify the location of the Application Engine trace file; it applies only to the generic PeopleTools SQL and PeopleCode traces.

Setting Options in PeopleSoft Configuration Manager

For processes running on a Microsoft Windows workstation, you can set trace options using PeopleSoft Configuration Manager. This procedure is valid only if you are running Application Engine programs on a Microsoft Windows workstation—the development environment.



Application Engine Trace check boxes

To set Application Engine traces:

1. Start Configuration Manager, and select the Trace tab.
2. Select appropriate trace options.

You can select any combination of the options.

3. Click either the Apply or OK buttons to set trace options.

Locating Trace Files

Where you look for the generated trace file depends on how you invoked the program and the operating system on which the program runs, as shown in the following table:

<i>Location Where the Program Was Initiated</i>	<i>Trace File Location</i>
Microsoft Windows workstation	Look for the trace file in %TEMP%\PS\<db name>.
PeopleCode	Look for the trace file in %TEMP%\PS\<db_name> on Microsoft Windows NT and in PS_CFG_HOME /log/<db name> on UNIX and Linux systems.

<i>Location Where the Program Was Initiated</i>	<i>Trace File Location</i>
Command line	Look for the trace file in the directory specified in the Log/Output field in the PS_SERVER_CFG file.
PeopleSoft Process Scheduler	Look for the trace file in a subdirectory of the directory specified in the Log/Output field in the PS_SERVER_CFG file.

When a program includes a process instance, Application Engine names the trace file according to the following convention: *AE_Program_name_Process_Instance.AET*. When the program does not include a process instance, Application Engine names the file according to this convention: *AE_Date/Time_Stamp_OS_PID.AET*. The date and time stamp is in the format month, day, hour, minute, second, with two values for each date element and no punctuation between elements. For example, August 12 at 5:09 p.m. and 30 seconds would be 0812170930.

Note. For an Application Engine program running on a server, PeopleTools writes the generic PeopleTools trace for SQL and PeopleCode trace files to the same directories as the AET traces. The prefix of the trace file name is also the same, and the suffix is *trc*. On the Windows workstation, the trace is written to the file specified in the People Tools Trace File field on the Trace tab of PeopleSoft Configuration Manager.

Chapter 9

Using Temporary Tables

This chapter provides an overview of temporary tables and discusses how to:

- Create temporary table instances.
- Manage temporary table instances.
- Make external calls.
- View temporary table usage.

Understanding Temporary Tables

Because Application Engine programs run in batch mode, multiple instances of the same program often execute in parallel. When this happens, there is a significant risk of data contention and deadlocks on tables. To avoid this, you can dedicate specific instances of temporary tables for each program run.

You can also use temporary tables to improve performance. For example, if you find that, multiple times during a run, the program accesses a small subset of rows from a much larger table, you can insert the necessary rows into a temporary table as an initialization task. Then the program accesses the data residing in the smaller temporary table rather than the large application table. This technique is similar to reading the data into an array in memory, except that the data never leaves the database, which is an important consideration when the program employs a set-based processing algorithm.

Any number of programs, not just Application Engine programs, can use the temporary table definitions. When you specify a temporary table on the Temp Tables tab in the Application Engine program properties, Application Engine automatically manages the assignment of temporary table instances. When Application Engine manages a dedicated temporary table instance, it controls the locking of the table before use and the unlocking of the table after use.

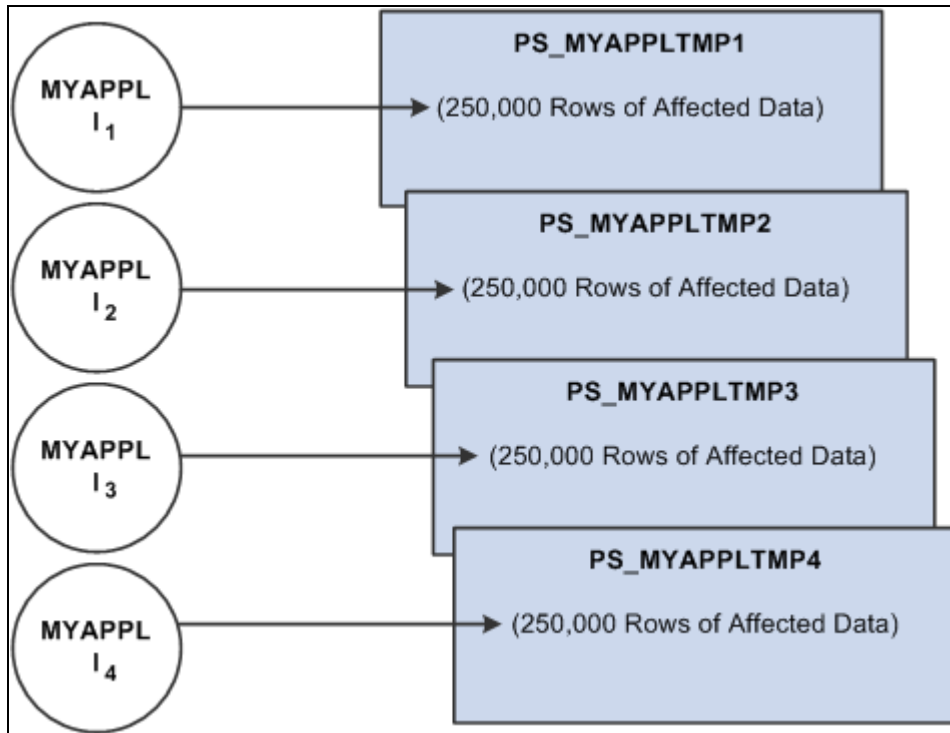
Parallel Processing

Parallel processing is used when considerable amounts of data must be updated or processed within a limited amount of time, or batch window. In most cases, parallel processing is more efficient in environments containing multiple CPUs and partitioned data.

To use parallel processing, partition the data between multiple concurrent runs of a program, each with its own dedicated version of a temporary table (for example, PS_MYAPPLTMP). If you have a payroll batch process, you could divide the employee data by last name. For example, employees with last names beginning with A through M get inserted into PS_MYAPPLTMP1; employees with last names beginning with N-Z get inserted into PS_MYAPPLTMP2.

To use two instances of the temporary table, you would define your program (say, MYAPPL) to access to one of two dedicated temporary tables. One execution would use A-M and the other N-Z.

The Application Engine program invokes logic to pick one of the available instances. After each program instance is matched with an available temporary table instance, the %Table meta-SQL construct uses the corresponding temporary table instance. Run control parameters passed to each instance of the MYAPPL program enable it to identify which input rows belong to it, and each program instance inserts the rows from the source table into its assigned temporary table instance using %Table. The following diagram illustrates this process:



Multiple program instances running against multiple temporary table instances

No simple switch or check box enables you to turn parallel processing on and off. To implement parallel processing, you must complete the following set of tasks. With each task, you must consider details about your specific implementation.

1. Define and save temporary table records in Application Designer.

You do not need to run the SQL Build process at this point.

2. In Application Engine, assign temporary tables to Application Engine programs, and set the instance counts dedicated for each program.

Employ the %Table meta-SQL construct so that Application Engine can resolve table references to the assigned temporary table instance dynamically at runtime.

3. Set the number of total and online temporary table instances on the PeopleTools Options page.
4. Build temporary table records in Application Designer by running the SQL Build process.

Creating Temporary Table Instances

This section provides an overview of temporary table instances and discusses how to:

- Define temporary tables.
- Set the number of temporary table instances.
- Build table instances.

Understanding Temporary Table Instances

To run processes in parallel, you need to enable multiple instances of the same temporary table. You use the PeopleTools Options page to set the number of temporary table instances for Application Engine processes started online from the PeopleCode CallAppEngine function.

This global setting is separate from the instance count setting for a particular program. To use a temporary table with a specific program, you assign the table to the program and set the number of instances created when a particular program is run.

Key Fields for Temporary Tables

To take advantage of multiple instances of a temporary table, use the Temporary Table record type.

Insert the PROCESS_INSTANCE field as a key on any temporary tables that you intend to use with Application Engine. Application Engine expects Temporary Table records to contain the PROCESS_INSTANCE field.

Note. When all instances of a temporary table are in use and the Continue runtime option on the Program Properties dialog box Temp Table tab is selected, PeopleTools inserts rows into the base table using PROCESS_INSTANCE as a key. If you do not include PROCESS_INSTANCE as a key field in a temporary table, select the Abort Temp Table tab runtime option.

Temporary Table Performance Considerations

When you run batch processes in parallel, there is a risk of data contention and deadlocks on temporary tables. To avoid this, Application Engine has a feature that enables you to dedicate specific instances of temporary tables for each process. When Application Engine manages a dedicated temporary table instance, it controls the locking of the table before use and the unlocking of the table after use.

When you decide on the number of instances for temporary tables for a process, you must take into consideration the number of temporary tables that the process uses. The more instances you have the more copies of the temporary tables you will have on your system. For example, if a process uses 25 temporary tables and you have 10 instances for a process, you will have 250 temporary tables on your system.

On the other hand, if you are running a process in parallel and all of the dedicated temporary table instances are in use, this slows down the performance of the process. So, you will need to find a balance that works for your organization.

If you need more temporary table instances after you have entered production, you must rebuild all of your temporary tables so that the database reflects the proper inventory of instances. While the build process runs, users cannot access the database. Because of this, spend time deriving adequate estimates as to the number of temporary tables required.

A physical table within the database, named PS_AEONLINEINST, stores online temporary table instance usage. If you notice performance issues related to online Application Engine program runs, enable the Application Engine SQL and Timings trace.

If the following SQL command requires more time than normal to complete, this is a good indication that not enough online temporary instances are defined on the PeopleTools Options page.

```
UPDATE PS_AEONLINEINST . . .
```

Defining Temporary Tables

To define a temporary table:

1. In Application Designer, select File, New.
2. Select Record from the New Definition dialog box.
3. Select Insert, Field, and insert the PROCESS_INSTANCE field.
4. Select the Record Type tab and select the Temporary Table option.

Setting the Number of Temporary Table Instances

Select PeopleTools, Utilities, Administration, PeopleTools Options to access the PeopleTools Options page.

PeopleTools Options

Environment Long Name: Environment Short Name:

System Type:

Language Settings

Language Code: *Sort Order Option:

☐ Translations Change Last Update

Background Disconnect Interval:

☐ Multi-Company Organization

☒ Multi-Currency

☒ Use Business Unit in nVision

☒ Use Secure Rep Rqst in nVision

☐ Multiple Jobs Allowed

☒ Allow DB Optimizer Trace

☒ Grant Access

☒ Platform Compatibility Mode

☐ Allow NT batch when CCSID<>37

☐ Save Error is Fatal

☐ Set Focus on Save Button

Temp Table Instances (Total):

Temp Table Instances (Online):

*Maximum Message Size:

Base Time Zone:

Last Help Context # Used:

*Data Field Length Checking:

*Maximum Attachment Chunk Size:

Upgrade Project Commit Limit:

*Enable Switch User:

*Case Insensitive Searching:

Style Sheet Name:

Branding Application Package:

Branding Application Class:

Max rows in search results:

Default rows in search results:

Tree Manager Options

☐ Use Tree Update Reservation

Max Tree Inactivity Period,min:

Help Options

F1 Help URL:

Ctrl-F1 Help URL:

WSRP Display Mode

WSRP Display Mode:

Database Encryption Algorithm

Database Encryption Algorithm:

PeopleTools Options page

The system determines the total available number of temporary table instances for a base table according to the settings for total and online instances that you make on this page.

Temp Table Instances (Total) (temporary table instances [total])	The difference between the total and online numbers is your EPM-managed tables. If you are not using PeopleSoft EPM, the total and online numbers should be the same.
Temp Table Instances (Online) (temporary table instances [online])	<p>Enter the number of temporary table instances for Application Engine processes started online from the PeopleCode CallAppEngine function. In general, the number you enter should be relatively small (less than 10), so that extra instances do not affect performance.</p> <p>Application Engine uses this value to identify a range of temporary tables devoted to programs called by the CallAppEngine function. A randomizing algorithm balances the load for the online process that is assigned to a temporary table devoted to online program execution.</p>

Building Table Instances

The system builds temporary table instances at the same time it builds the base table for the record definition. When the system builds a table (as in, Build, Current Object) and the record type is Temporary Table, it determines the total number of instances of the temporary table based on the settings that you made on the PeopleTools Options page.

The system creates a maximum of 99 temporary table instances, even if the sum exceeds 99 for a particular temporary table.

The naming convention for the temporary table instances is as follows: BaseTableName_Number, where *Number* is a number between 1 and 99, as in PS_TEST_TMP23.

Note. You can take advantage of database-specific features such as table spaces and segmentation. For instance, you may want to use the Build process to generate a data definition language (DDL) script, and then fine-tune the script before its execution, or you could place different sets of temporary tables on different table spaces according to instance number.

Managing Temporary Table Instances

This section provides an overview of temporary table instance numbers and discusses how to:

- Assign temporary tables to programs.
- Adjust meta-SQL.

Understanding Temporary Table Instance Numbers

You use the Temp Tables tab in the Program Properties dialog box to manage the number of different batch or dedicated temporary tables required for each program definition and the number of instances of each. You select all the necessary temporary table records to meet the needs of your program's logic.

Note. You must set the instance count on the Temp Tables tab before building the tables in Application Designer.

Regardless of the instance counts value in the Application Engine program properties or on the PeopleTools Options page, make sure that you have the appropriate records assigned to the appropriate programs. You also need to ensure that the SQL inside your Application Engine program contains the correct usage of the %Table construct.

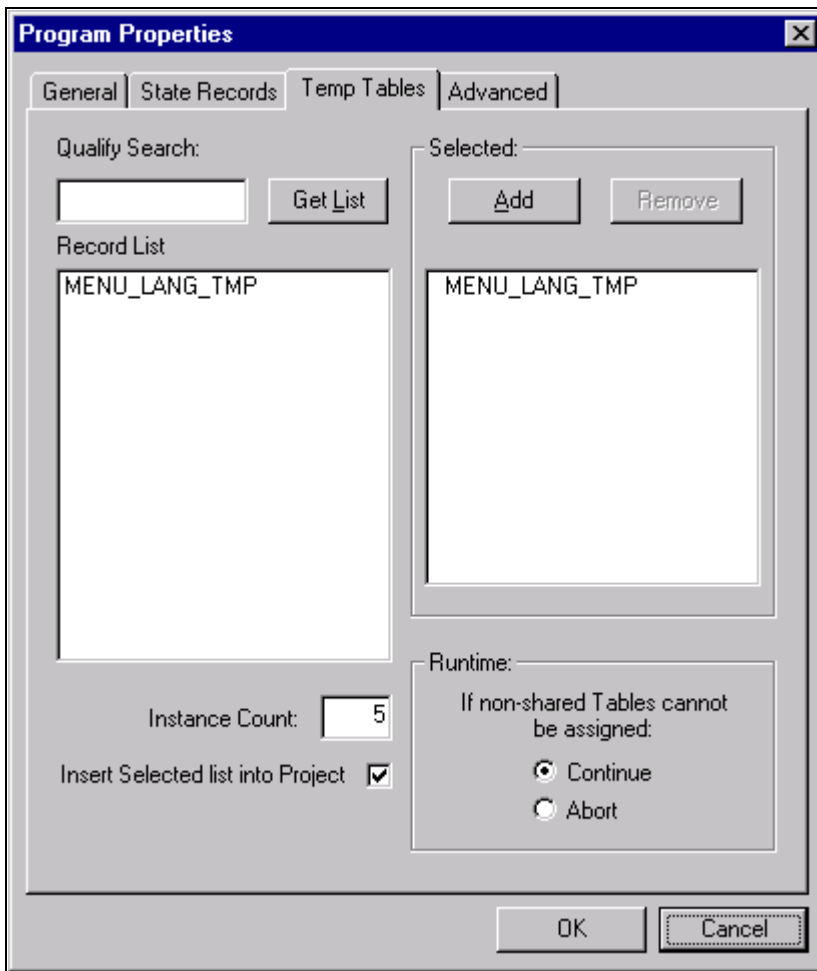
The number of temporary table instances built for a specific temporary table record during the SQL Build process is the value of the total temporary table instances from the PeopleTools Options page added to the sum of all the instance count values specified on the Temp Table tab for the Application Engine programs that use that temporary table.

For example, assume that we have defined APPLTMPA as a temporary record type. If the number of total temporary table instances is set to 10, and APPLTMPA appears in the Temp Tables tab in the Program Properties dialog box for two Application Engine programs. In one program, the instance count is set to 3, and in the other, the instance count is set to 2. When you run the SQL Build process, PeopleTools builds a total of 15 temporary table instances for APPLTMPA.

The total and online instance counts should be equal, unless your PeopleSoft application documentation provides specific instructions on setting these values differently. When the values are equal, the Temp Table Instances (Total) field controls the number of physical temporary table instances to be used by online programs that Application Designer creates for a temporary table record definition. If the value for the Temp Table Instances (Online) field is less than the value for the Temp Table Instances (Total) field, the difference between the two numbers provides a pool of tables for backward compatibility for developers who took advantage of the %Table(record_name, instance_number) approach for manually managing temporary table locking, (such as PeopleSoft EPM).

Assigning Temporary Tables to Programs

Open an Application Engine program in Application Designer. Select File, Definition Properties. Select the Temp Tables tab.



Program Properties dialog box: Temp Tables tab

In the Record List box, include all the necessary temporary table records for this program.

In the Instance Count field, specify the number of copies of temporary tables for a program. Anytime you change the instance counts, you need to rebuild the temporary tables to ensure that the right number of instances get created and are available for your programs.

Note. The concept of dedicated temporary tables is isolated to the Application Engine program run. The locking, truncate/delete from, and unlocking are designed to occur within the bounds of an Application Engine program run. Therefore, the system does not keep a temporary table instance available after the Application Engine program run is over.

Runtime Allocation of Temporary Tables

Online processes have their own set of dedicated temporary tables, defined globally on the PeopleTools Options page. When you invoke a process online, PeopleTools randomly allocates a single temporary table instance number to a program for all its dedicated temporary table needs. While the program runs, no other program can use that instance number. Any other online process that happens to get the same instance value waits for the first program to finish, so that the instance number is unlocked.

In contrast, batch processes are allocated temporary table instances on a record-by-record basis. The system begins with the lowest instance number available for each temporary table until all of the temporary table instances are in use. If there are not any temporary tables, available and you selected *Continue* for the *If non-shared Tables* cannot be assigned group box, then the base table is used, with the process instance number as a key.

When a program ends normally or is cancelled with Process Monitor, the system automatically releases the assigned instances.

Condition	Online	Batch
Temporary tables are allocated using meta-SQL.	%Table(temp-tbl)	%Table(temp-tbl)
Temporary tables are allocated at runtime.	Psae.exe randomly assigns an instance number from the number range on your online temporary table setting on the PeopleTools Options page. Psae.exe uses that number for all tables for that program run.	Individually allocates an instance number based on availability on a record-by-record basis. Psae.exe begins with the lowest instance number available for each temporary table, until all of the instances are in use.
No temporary tables are free.	For a particular record, if the instance is currently in use and the program is set to <i>Continue</i> , then the psae.exe queues the program until the assigned instance number becomes free.	If the program is set to <i>Continue</i> , the system uses a shared base table. If the program is set to <i>Abort</i> , then the system terminates the program. <i>Never</i> queues for a table.
A temporary table is initially clear.	Yes, when program instance becomes available.	Yes, when assigned.
An instance number is locked.	The lock is on when the program is loading into memory.	The lock is on when the program is loading into memory. For restartable programs, the temporary tables remain locked across restarts until the program has completed successfully or until the temporary tables are manually released using Process Monitor or the Manage Abends page.

Condition	Online	Batch
An instance number is unlocked.	Temp tables unlocked on completion of program. In the event of a kill or a crash, the tables remain locked, and the tables must be freed using Process Monitor or the Manage Abends page.	If restart is disabled, the temporary tables are unassigned automatically in the event of a controlled abnormal termination. If you cancel a process using Process Monitor, PeopleTools frees the temporary tables automatically. When you use the Manage Abends page, you must click the Temp Tables button corresponding to the correct process instance, and then click the Release button on the Temporary Tables tab of the Application Engine program properties.

Note. When you have manually released the temporary tables from their locked state, you lose any option to restart the program run.

Sharing Temporary Table Data

Dedicated temporary tables do not remain locked across process instances. If sequential Application Engine programs need to share data by way of temporary tables, a parent Application Engine program should call the programs that share data.

Adjusting Meta-SQL

A critical step in implementing parallel processing is to make sure that you have included appropriate meta-SQL within the code that your Application Engine program executes.

Referencing Temporary Tables

To reference a dedicated temporary table, you must use:

```
%Table(record)
```

You can reference any table with %Table, but only those records defined as temporary tables get replaced with a dedicated instance table by Application Engine. When you are developing programs that take advantage of %Table, choose temporary table indexes carefully. Depending on the use of the temporary table in your program and your data profile, the system indexes may be sufficient. On the other hand, a custom index may be needed instead, or perhaps no indexes are necessary at all. Consider these issues when designing your application. You want to define indexes and SQL that perform well in most situations, but individual programs or environments may require additional performance tuning during implementation.

Note. The default table name refers to PS_recname, where PS_recname1,2,... represents the dedicated temporary tables.

As Application Engine resolves any %Table, it checks an internal array to see if a temporary table instance has already been chosen for the current record. If so, then Application Engine substitutes the chosen table name. If not, as in when a record does not appear in the temp table list for the program, then Application Engine uses the base table instance (PS_recname) by default. Regardless of whether %Table is in PeopleCode SQL or in an Application Engine SQL Action the program uses the same physical SQL table.

Populating the Temporary Table Process Instance with the Process Instance

All temporary tables should be keyed by process instance. If you use the Continue option when batch or dedicated tables cannot be assigned, Process Instance is required as a key field. The current process instance is automatically put into the state record, but when you insert rows into your temporary tables, you must supply that process instance. Use %ProcessInstance or %Bind(PROCESS_INSTANCE) meta-SQL to return the numeric (unquoted) process instance.

The process instance value is always zero for programs initiated with the CallAppEngine function. This is because the program called with CallAppEngine runs in process; that is, it runs within the same unit of work as the component with which it is associated.

If you are using dedicated tables and have elected to continue if dedicated tables cannot be assigned, then SQL references to dedicated temporary tables must include PROCESS_INSTANCE in the Where clause.

Clearing Temporary Tables

You do not need to delete data from a temporary table manually. The temporary tables are truncated automatically when they are assigned to your program. If the shared base table has been allocated, because no dedicated instances were available, then Application Engine performs a delete by process instance instead of performing a truncate. In such a case, PROCESS_INSTANCE is required as a high-level key.

You can perform additional deletes of temporary table results during the run, but you must include your own SQL action that uses the %TruncateTable function. If the shared base table has been allocated because no dedicated instances were available, then %TruncateTable is replaced with a delete by process instance instead of a truncate.

Note. You should always use %TruncateTable to perform a mass delete on dedicated temporary tables, especially if the Continue option is in effect.

Even if you have elected to terminate the program if a dedicated table cannot be allocated, you may still use %TruncateTable meta-SQL with dedicated temporary tables. %TruncateTable resolves to either a Truncate or a Delete by process instance, as needed.

The argument of %TruncateTable is a table name instead of a record name. As a result, you must code your SQL as shown in this example:

```
%TruncateTable(%Table(recname))
```

Note. You should avoid hard-coded table names inside %TruncateTable, since they preclude the possibility of concurrent processing.

Making External Calls

When you call one Application Engine program from another, the assignment of dedicated tables for the called, or child, program, occurs only if the calling, or parent, program is in a state where a commit can occur immediately.

PeopleTools enables you to commit immediately, so that Application Engine can commit the update it performs to lock the temporary table instance. Otherwise, no other parallel process could perform any assignments. In general, this means that you should issue a commit just prior to the Call Section action.

While making external program calls, note the following:

- If the situation is suitable for a commit, then the temporary table assignment and the appropriate truncates occur.
- If the situation is not suitable for a commit, and the called program is set to continue if dedicated tables cannot be allocated, then the base tables are used instead, and a delete by process instance is performed.
- If the situation is not suitable for a commit and the called program is set to terminate if dedicated tables cannot be allocated, then program execution terminates.

This situation reflects an implementation flaw that you need to correct.

- If the called Application Engine program shares temporary tables with the calling program, this is allowed.

Common temporary tables are the way you share data between the calling and called programs. Application Engine locks only instances of temporary tables that have not already been used during the current program run. Temporary tables that already have an assigned instance continue to use that instance.

External Calls in Batch Mode

For batch runs, list in the program properties of the root program all of the temporary tables that any called programs or sections use. This ensures that the tables get locked sooner and as a single unit. This approach can improve performance, and it ensures that all the tables required by the program are ready before execution starts.

External Calls in Online Mode

If the online program run is designed to use any temporary tables at any point during the CallAppEngine unit of work, then the root program must have at least one temporary table specified in the Application Engine program properties. This is true even if the root program does not use temporary tables. This is required so that the system locks the instance number early on to avoid an instance assignment failure after the process has already started processing.

All temporary tables used by a specific program, library, or external section must be specified in that program to ensure that the system issues truncates (deletes) for the tables being utilized.

If no temporary tables appear in the root program properties, and Application Engine encounters a %Table reference for a temporary table record, an error appears.

Sample Implementation

The following scenario describes the runtime behavior of Application Engine and temporary tables.

Assume you have Program A and Program B, and three temporary table definitions: PS_TMPA, PS_TMPB, and PS_TMPC. Values on the Temporary Tables tab in the Program Properties dialog box for each program are as follows:

- Program A: PS_TMPA and PS_TMPB are specified as the dedicated temporary tables, and the instance count is 4.
- Program B: PS_TMPB and PS_TMPC are specified as the dedicated temporary tables, and the instance count is 3.

After you run the SQL Build process in Application Designer, the following inventory of temporary tables appears in the database.

For PS_TMPA:

- PS_TMPA1
- PS_TMPA2
- PS_TMPA3
- PS_TMPA4

For PS_TMPB:

- PS_TMPB1
- PS_TMPB2
- PS_TMPB3
- PS_TMPB4
- PS_TMPB5
- PS_TMPB6
- PS_TMPB7

For PS_TMPC:

- PS_TMPC1
- PS_TMPC2
- PS_TMPC3

Because the instance count for Program A is 4, the system builds four instances of PS_TMPA and PS_TMPB for Program A to use. Because the instance count for Program B is 3, the system builds an additional three instances of PS_TMPB and three instances of PS_TMPC for Program B to use.

Notice that because Program A and Program B are sharing PS_TMPB, there are seven instances. The system derives this total by adding the instance count value from all the programs that share a particular temporary table instance. In this case, the four from Program A and the three from Program B combine to require a total of seven instances of PS_TMPB to be built.

Given that this collection of temporary tables exists in your database, let's say that you start Program A. At runtime, Application Engine examines the list of temporary tables dedicated to Program A, and assigns the first available instances to Program A. Then, assuming that no other programs are running, Application Engine assigns PS_TMPA1 and PS_TMPB1 to Program A.

Suppose that shortly after you started Program A, another user started Program B. Again, Application Engine examines the list of temporary tables dedicated to Program B and assigns the first available instances. In this scenario, Application Engine assigns PS_TMPB2 and PS_TMPC1 to Program B. Because Program A is already using PS_TMPB1, the system assigns PS_TMPB2 to Program B.

The system assigns records, such as TMPA, to programs. The base tables, such as PS_TMPA, are also built, by default, in addition to the dedicated temporary instances. If the Program Properties dialog box setting for the Temp Tables tab is set to Continue when no instances are available, the system uses the base table instead of the dedicated instance.

Viewing Temporary Table Usage

This section discusses how to:

- View temporary table usage by record.
- View temporary table settings by program.
- View online instance usage.
- Resolve the temporary table usage warning message.

Viewing Temporary Table Usage by Record

Select PeopleTools, Application Engine, Review Temp Table Usage to access the Temp Table Usage by Record page.

Temp Table Usage by Record

Temp Table Settings by Program

Temp Table Usage by Record

Filter List by

Record (Table) Name:

Program Name:

Refresh

Lock Details

Record (Table) Name	Program Use Count	Total Instances	Locked Instances	Unused Instances	View Programs
AEEXT_TAO	1	5	0	5	View Programs
PSMSFTMPCOM	1	10	0	10	View Programs
PSMSFTMPIDS	1	10	0	10	View Programs
PSMSFTMPTBL	1	10	0	10	View Programs
PTPP_CPKP_TMP	1	1	0	1	View Programs
QE_AEEXT_TAO	1	5	0	5	View Programs
QE_AETEST_TAO	2	10	0	10	View Programs

Temp Table Usage by Record page

If you implemented temporary tables for parallel Application Engine program runs, use this page and the Temp Table Usage by Program page to find out how the system allocates temporary tables to your programs.

Parallel processing is designed to be a performance enhancing option. However, if the demand for temporary table instances consistently exceeds the current supply, performance suffers. Also, in other situations, your inventory of temporary table instances may far outnumber demand. Here, you may consider reducing the number of instances provided to conserve system resources.

This page shows you the following metrics for evaluating inventory and allocation of temporary tables.

Program Use Count	Shows the instance count of listed program.
Total Instances	Shows the total number of existing instances of a temporary table.
Locked Instances	Shows the current number of instances that they system has locked for program runs.
Unused Instances	Shows the current number of instances that are available for use.

Viewing Temporary Table Settings by Program

Select PeopleTools, Application Engine, Review Temporary Table Usage, Temp Table Settings by Program to access the Temp Table Settings by Program page.

Temp Table Usage by Record

Temp Table Settings by Program

Temp Table Settings by Program

Filter List by

Record (Table) Name:

Program Name:

Refresh

Settings Details

Program Name	Total Instances	Batch Only	Abort Flag	Disable Restart	View Records
AETESTEXT	5	N	N	N	View Records
PSMSFGENSEQ	10	N	Y	Y	View Records
PTPP_CPPROJ	1	N	N	N	View Records
QE_AERESTART	5	N	N	N	View Records
QE_AETESTEXT	5	N	N	N	View Records
QE_AETESTPRG	5	N	N	Y	View Records

Temp Table Settings by Program page

If the Application Engine process was started in Process Monitor, you can select PeopleTools, Application Engine, Manage Abends to access the Manage Abends page and then the Process Monitor.

Viewing Online Instance Usage

Select PeopleTools, Application Engine, Review Online Instance Usage to access the Online Instance Usage page.

Online Instance Usage		Reset Counts to 0
Locks Issued by Instance		
Temp Table Instance	Number of Locks Issued	
0	0	
1	0	
2	0	
3	0	
4	0	

Online Instance Usage page

Resolving the Temporary Table Usage Warning Message

If an Application Engine batch program is unable to get a dedicated temporary table because all instances are locked, but it can use the base table, the system issues a warning. However, if the program has been set to terminate when a dedicated instance is not available, then the program terminates even if the base table can be used.

You could see the warning message in two ways:

- A warning message appears in the standard output of the process.

When running from the command prompt, the message appears in that window. When the program is running on a server through PeopleSoft Process Scheduler, the output is sent to the standard status file, which you can access using Process Monitor.

- A warning message appears in the AET trace file if a dedicated temporary table instance cannot be locked because none are available.

This message appears in the trace file regardless of the trace settings you have selected.

If you see the warning regarding base temporary table usage, this means either there aren't enough temporary table instances defined or some locked instances that must be released.

When a restartable process terminates abnormally, the temporary tables stay locked to enable a smooth restart. However, if you do not want to restart the process, then the locked temporary tables must be released. When you cancel the process using Process Monitor, the release of locked temporary tables occurs automatically. If the process was not launched through PeopleSoft Process Scheduler, Process Monitor does not track the process. Because of this, you must use the Manage Abends page to release temporary tables used by processes invoked outside of PeopleSoft Process Scheduler.

Index

Symbols

%ListBind
 using 96
%ResolveMetaSQL meta-SQL function 90, 99

A

action
 Call Section 34
actions
 Do Select 39
 Do Until 39
 Do When 38, 71
 Do While 39
 execution order 34, 61
 inserting 35
 setting properties 35
 specifying call section 41
 specifying Do 38
 specifying log message 42
 specifying PeopleCode 40
 specifying SQL 37
 specifying XSLT 43
 understanding 6, 34
 understanding call section 8
 understanding Do 7
 understanding implementation phase for
 specifying 3
 understanding log message 7
 understanding PeopleCode 7
 understanding SQL 7
active status 18
AERSection class 66
APIs, calling PeopleTools 70
Application Designer 16
Application Engine 4, 5
 caching the server 136
 controlling abnormal terminations 134
 enabling the Debugger 126
 enabling traces 156
 introducing *See* Application Engine
 meta-SQL *See Also* meta-SQL, meta-SQL
 PeopleCode *See Also* PeopleCode
 programs
 See Also Application Engine programs
 requests 121
 reusing SQL 48
 set processing 50
 specifying actions 3
 understanding xi, 1, 5
 understanding implementation 2
Application Engine, understanding
 implementation phase for setting up properties
 2
Application Engine program elements 6, 8
application engine programs
 tracing 147
Application Engine programs

 accessing properties 23
 adding sections 28
 AEDAEMONMGR 9
 assigning temporary tables 167
 calling from COBOL 139
 calling PeopleTools APIs 70
 copying/moving elements 21
 creating, opening and renaming 19
 creating process definitions 118
 daemon 9
 debugging 126
 environment 63
 executing manually via command line 118
 executing online via PeopleCode 118
 execution options 117
 freeing locked temporary tables 137
 including dynamic SQL 74
 inserting actions 35
 inserting sections 29
 inserting steps 31
 invoking via command line 123
 invoking via PeopleCode 122
 listing process definitions parameters 120
 locating sections 29
 making synchronous online calls to 66
 managing 117
 printing 19
 restarting 131
 running 117
 setting action properties 35
 setting advanced properties 26
 setting commits 47
 setting general properties 23
 setting section properties 30
 setting state record properties 23
 setting step properties 32
 shared values in COBOL programs 68
 specifying actions 33
 specifying temporary tables 25
 starting parallel processing 10
 starting via Process Request page 121
 testing 21
 tracing 147
 transform 10
 types 8
 understanding 6, 11
 using CommitWork 71
 using PeopleCode 59
 using PeopleCode (examples) 71
 variables 60
 viewing 11
application library 27
application servers
 enabling traces in configuration files 157
 running batch programs 118
arrays 73

B

BEA Tuxedo 118

- beginning 135
- behavior 112
 - filtering section 17
- bind variables 77
- built-ins
 - peoplecode 150
- bulk insert 36, 50
- Bulk Insert statement 50

C

- cache
 - setting parameters 137
- caching 136
- caching, Application Engine server 136
- calls
 - batch/online mode 172
 - calling COBOL modules 67
 - calling PeopleTools APIs 70
 - calling programs from COBOL 139
 - calling programs via PeopleCode 122
 - calling sections in other programs 46
 - making external calls 172
 - making synchronous online calls to programs
 - 66
 - specifying call section actions 41
 - understanding call section actions 8
 - using RemoteCall 69, 123
- call section references, finding 29
- COBOL 114
 - adding copybooks to COBOL programs 139
 - assigning copybook values 141
 - calling modules 67
 - calling programs 139
 - handling errors 145
 - transferring data between COBOL and
 - Application Engine programs 140
- command line
 - executing manual programs 118
 - invoking programs 123
 - options 124
 - restarting programs 134
 - starting PSDAEMON 9
 - tracing programs 156
- command line syntax 9
- comments
 - show 13
- commits
 - calling COBOL modules 69
 - making external calls 172
 - setting commit levels for sections 31
 - setting commit levels for steps 32
 - setting commits for programs 47
 - understanding restarts 40, 131
 - using the CommitWork function 71
 - using variables 60
- Commit statements 50
- CommitWork 71
- component variables
 - peoplecode 61
- Configuration Manager
 - enabling/disabling restart 136
 - enabling the Application Engine Debugger
 - 126
 - setting trace options 158
- considerations

- no rows setting 48
- Considerations
 - database 113
- constructs
 - %BINARYSORT 76
 - %Bind 76
 - %DateIn 83
 - %DateTimeIn 85
 - %EffDtCheck 86
 - %InsertSelect 91
 - %InsertSelectWithLongs 91
 - %Join 91
 - %Like 92
 - %LikeExact 92
 - %List 93
 - %ListBind 95
 - %ListEqual 96
 - %NoUpperCase 98
 - %NumToChar 99
 - %Select 103
 - %SelectInit 104
 - %SQL 105
 - %Table 107
 - %Test 107
 - %TextIn 108
 - %TimeAdd 108
 - %TimeIn 108
 - %TimeOut 109
 - %TruncateTable 110
 - %UpdateStats 111
 - %Upper 115
 - understanding 59
- copybooks
 - adding to COBOL programs 139
 - assigning values 141
- creating 11
- creating new programs 19
- custom 17

D

- daemon program 9
- database 156
- database function 112
- database record 61
- databases
 - database optimizer trace 153
 - improving performance 55
- DB2 OS/390
 - tracing 155
 - using %UpdateStats 113, 114
- DB2 UNIX
 - tracing 155
 - using %UpdateStats 112
- debugger 126
- debugging
 - enabling the Application Engine Debugger
 - 126
 - programs 126
 - setting 127
- debugging options
 - setting up 4
- default 17
- definition 15
 - jump to 15
- Definition Filter dialog box 16

- filtering section 17
- Definition view
 - using 12
- disabling
 - %updatestats 114
- Do actions 38
- Do Until action 39

E

- errors
 - abnormal terminations 134
 - AEDAEMONMGR 9
 - bad restarts 135
 - calling PeopleTools APIs 70
 - handling COBOL errors 145
 - responding at the step level 33
 - specifying log message actions 42
 - triggering an error status 65
 - using SQL trace 151
- existing programs
 - opening 20

F

- file class 67
- filtering
 - set processing 54
 - views 16
- flows
 - printing definitions 19
 - program flow pop-up menu 14
 - using program flow view 13
- functions
 - %Abs 74
 - %Cast 78
 - %ClearCursor 79
 - %COALESCE 80
 - %DateAdd 83
 - %DateDiff 83
 - %DateOut 84
 - %DatePart 84
 - %DateTimeDiff 84
 - %DateTimeOut 85
 - %DecDiv 85
 - %DecMult 86
 - %DTTM 86
 - %Execute 87
 - %ExecuteEdits 87
 - %Mod 97
 - %Next 98
 - %Previous 98
 - %Round 102
 - %RoundCurrency 102
 - %Substring 106
 - %TimePart 109
 - %TrimSubstr 109
 - %Truncate 110
 - %TruncateTable 110
 - CallAppEngine 63, 66, 118, 122
 - CommitWork 71
 - Exit 65
 - math 72
 - RemoteCall 67, 123

- understanding 59
- fundamentals
 - Application Engine 5

G

- get list 25
- global variables
 - peoplecode 61

I

- implementation phases, understanding
 - implementation 2
- Informix
 - command line options 124
 - tracing 155
 - using %UpdateStats 113, 114
- insert
 - section into project 13
- instance count 26
- instances 163

J

- jump
 - program flow 13

L

- local variables
 - peoplecode 61
- location
 - cache directory 137
- logging
 - specifying log message actions 42
 - understanding log message actions 7
- Log Message action 42
- look option
 - example 130

M

- MAIN 6
- market 18, 30
- math functions 72
- menus
 - Definition view popup menu 12
 - Program Flow view popup menu 14
- message set 27
- meta-SQL
 - adjusting for temporary tables 170
 - constructs, functions, and meta-variables 74
 - understanding 5, 59
- meta-SQL functions
 - %ResolveMetaSQL 90, 99
- meta-variables

- %AeProgram 75
- %AeSection 75
- %AeStep 75
- %AsOfDate 75
- %AsOfDateOvr 75
- %Comma 80
- %Concat 80
- %CurrentDateIn 81
- %CurrentDateOut 81
- %CurrentDateTimeIn 81
- %CurrentDateTimeOut 82
- %CurrentTimeIn 82
- %CurrentTimeOut 82
- %DateNull 83
- %DateTimeNull 85
- %FirstRows 89
- %JobInstance 91
- %LeftParen 92
- %ProcessInstance 99
- %ReturnCode 101
- %RightParen 102
- %RunControl 103
- %Space 105
- %SQLRows 106
- %TimeNull 108
- understanding 59
- methods section 150
- Microsoft SQL Server *See* MS SQL Server
- modify option
 - example 131
- MS SQL Server
 - tracing 154
 - using %UpdateStats 113

N

- new programs
 - creating 20
- numbering, sequence 72

O

- Online Instance Usage page 176
- online mode
 - external calls 172
- online programs 118
- On Return 40
- opening existing programs 19
- optimizer trace
 - performance 156
- option 130, 131
- options 127
 - debugging 4, 128
 - section filtering 17
- Oracle
 - tracing 153
 - using %UpdateStats 112, 113
- overview 1
- overview of program types 8

P

- parallel processing
 - adjusting meta-SQL 170
 - set processing 54
 - understanding 161
 - using PSDAEMON 10
- PeopleCode 5
 - accessing state records 64
 - action execution order 61
 - AESession class 66
 - arrays 73
 - calling COBOL modules 67
 - deciding when to use 62
 - Do When actions 71
 - dynamic SQL 72
 - file class 67
 - invoking programs 122
 - making synchronous online calls 66
 - math functions 72
 - opening PeopleCode Editor 13, 14
 - PeopleCode sections in trace file 149
 - program environment 63
 - rowsets 72
 - sequence numbering 72
 - specifying actions 40
 - SQL class 73
 - understanding 60
 - understanding actions 7
 - using if/then logic 65
 - using in loops 65
 - using in programs (examples) 71
 - WINWORD 71
- PeopleSoft Configuration Manager
 - See* Configuration Manager
- PeopleSoft Integration Broker 7
- PeopleSoft Process Monitor *See* Process Monitor
- PeopleSoft Process Scheduler
 - See* Process Scheduler
- PeopleTools APIs, calling 70
- PeopleTools Options page 164
- platforms
 - filtering views 18
 - set processing issues 57
 - setting section properties 30
 - using database optimizer trace 153
- pop-up menus
 - Definition view 12
 - Program Flow view 14
- printing program/flow definitions 19
- Process & Definition page
 - definitions 118
- Process Definition Options page
 - listing 120
- Process Monitor
 - freeing locked temporary tables 137
 - locating program information 123
 - viewing batch timings 153
- ProcessRequest 10
- Process Request page
 - restarting programs 135
 - starting programs 121
- process scheduler 118
- Process Scheduler
 - enabling traces in configuration files 157
 - invoking BATTIMES.SQR 152
 - program execution options 117
 - restarting programs 135
 - running PSDAEMON 9
 - viewing batch timings 153

- program
 - command line 134
 - Process Request page 135
- program elements 6
- program flow view
 - switching 15
- Program Flow view 13
- program flow views 15
- program properties
 - called sections 42
 - setting 23
- Program Properties dialog box 167
- programs
 - renaming 20
- programs, Application Engine
 - See* Application Engine programs
- program type 10, 28
- program types 9
 - list of 8
- properties
 - setting up *See* Application Engine
- property
 - do select type 40
 - dynamic 41
 - no rows 37
 - program ID 41
 - section name 41
- PSDAEMON 9
- ptpecobl program 67

Q

- qualify search 25

R

- record list 25
- refreshing views 16
- refresh views
 - reordering definition objects 13, 15
- renaming programs 19
- restart
 - using 132
- Restartable option 48
- restarting 9
- restarting program
 - bad restart error 135
 - enabling/disabling 136
 - program-level 132
 - section-level 133
 - starting programs from the beginning 135
 - understanding 134
- restart program
 - understanding 131
- ReUse 36, 49
- rowsets 72
- Run Request dialog box 21
- runtime 26, 168

S

- section 151

- insert 13
- section name 30
- sections
 - execution order 28
 - finding 30
 - inserting 29
 - locating 29
 - section-level restarts 133
 - setting properties 30
 - understanding 6, 28
- section type 31
- sequence numbering 72
- set processing
 - avoiding row-by-row processing 53
 - examples 54
 - planning 51
 - platform issues 57
 - understanding 50
 - using 51
- Show All Comments 14
- SQL 5
 - expertise 51
 - including dynamic SQL 74
 - meta-SQL *See Also* meta-SQL
 - MS SQL Server *See Also* MS SQL Server
 - opening SQL Editor 13, 14
 - reusing statements 48
 - set processing 50
 - setting the ReUse Statement property 36
 - specifying actions 37
 - trace file SQL counts section 148
 - tracing 151
 - understanding actions 7
 - understanding dynamic 72
 - using bulk insert 50
 - using the SQL class 73
 - validating meta-SQL constructs 74
- SQL editor 13
- SQR 6, 45
- starting 10
- starting a daemon program
 - procedure 9
- statement
 - perform 7
 - select 7
- state records
 - accessing with PeopleCode 64
 - called programs 42
 - choosing record types 47
 - setting properties 23
 - sharing 46
 - understanding 8, 45
- step/action
 - insert 13
- steps
 - inserting 31
 - setting properties 32
 - step-level restarts 133
 - tracing 151
 - understanding 6, 31
- subqueries 55
- Sybase
 - command line options 124
 - using %UpdateStats 112
- syntax and parameters 68

T

- tables
 - Process Request page 121
 - set processing for denormalized tables 52
 - temporary *See Also* temporary tables
- techniques
 - avoiding 54
- temporary table
 - instances 3
 - performance 163
- temporary table instances
 - creating 3
- temporary tables 25
 - adjusting meta-SQL 170
 - allocating runtime 168
 - assigning to programs 167
 - building instances 166
 - calling other programs 172
 - clearing 171
 - creating instances 163
 - defining 164
 - external calls in batch/online mode 172
 - flattening 54
 - freeing locked 137
 - improving database performance 55
 - key fields 163
 - keying by process instance 171
 - managing instances 166
 - referencing 170
 - resolving the usage warning 176
 - set processing 52
 - setting the number of instances 164
 - sharing data 170
 - understanding 161
 - understanding instance numbers 166
 - understanding instances 163
 - using 161
 - viewing online instance usage 176
 - viewing settings by program 175
 - viewing settings by record 174
 - viewing usage 174
- terminations 134
- testing programs 21
- timings trace 151
- tips
 - debugging 128
- trace file
 - sections 148
- trace files 158
- trace results
 - understanding 147
- tracing
 - Application Engine programs 147
 - database optimizer 153
 - DB2 155
 - enabling 4
 - enabling Application Engine traces 156
 - environment information 151
 - Informix 155
 - locating files 158
 - MS SQL Server 154
 - Oracle 153
 - PeopleCode actions, built-ins and methods 149
 - program steps 151
 - setting command line options 156

- setting options in Configuration Manager 158
 - setting parameters in configuration files 157
 - SQL 151
 - SQL counts and timings 148
 - statement timings 151
 - summary data 150
- transforming program 10
- Tuxedo 118
- two-pass approach
 - using 54

U

- understanding 117
 - set processing 51
 - state records 45
- understanding actions 7
- understanding fundamentals 5
- understanding tracing 147
- UNIX 155
- using 115
 - %UpdateStats 114

V

- variables
 - Application Engine program 60
 - defining global 64
 - meta-variables *See Also* meta-variables
 - setting the cache directory 137
- view
 - menu 15
 - tabs 15
- view contents
 - filtering 16
- views
 - Definition 12
 - filtering 16
 - Program Flow 13
 - refreshing 16
 - switching between 15

W

- watch option
 - example 131
- work record 61

X

- XSLT
 - specifying actions 43
 - viewing 13