

[print](#) | [close](#)

Using PEX APIs to Trace Application-Specific Transaction Performance

[System iNEWS Magazine](#)

[Carsten Flensburg](#) [Keith Zblewski](#) [Keith Zblewski](#)

Carsten Flensburg

Thu, 12/01/2005 (All day)

[Click here to download code](#)

Those who develop or support 5250-based transaction applications running on the iSeries have many tools at their disposal for managing the performance of those applications. System commands such as WRKSYSSTS (Work with System Status) and WRKACTJOB (Work with Active Jobs) provide valuable information about key performance metrics such as transaction response time and transaction throughput.

Beyond the core operating system utilities, the reports in the Performance Tools licensed product or the Performance Management (PM) iSeries service offering let you track transaction performance metrics on a historical basis. When performance problems arise in an application of this kind, trace facilities invoked with the STRPFRTTC (Start Performance Trace) command can analyze individual transactions and identify causes of performance bottlenecks, such as a lock on a database record or program.

But what happens when a 5250-based application is changed or replaced with new technologies that use Web servers, application servers, and database technologies such as ODBC? The ability to analyze transaction performance becomes more complicated because tools such as WRKSYSSTS and the commonly used Performance Tools reports do not provide a measure of transaction performance for the non-5250 environment.

Fortunately, iSeries tools are available that can help you understand the mystery of application performance in this new world. It starts with a very powerful tool called the Performance Explorer (PEX). PEX has been available in OS/400 for many years, but few iSeries professionals have used it to help analyze application performance.

You can run PEX in several modes, with each mode providing a different view of performance or a different level of detail. The mode that you use to measure application transaction performance is called *trace mode*. Trace mode provides a way to trace various activities called *events* that occur when applications are running or system resources are being used. One special type of trace event is called a *user-defined transaction event*. Collecting user-defined transaction events with PEX is one way to measure the transaction performance of an application when a transaction is processed by multiple server jobs and/or threads.

For each transaction event, PEX captures the response time, CPU time, database and non-database I/O activity, and the seize/lock activity. This data can help a performance analyst understand how each transaction is performing and whether transactions are impeded by lock contention between multiple users of this application or other applications.

Before you can take advantage of the capability provided by PEX transaction events, you must follow two steps:

1. Define the application transactions that you want to track and measure.
2. Place transaction API calls in the application for the transactions you define in Step 1.

Defining Application Transactions

A *transaction* is any unit of work in your application that you want to track and measure. You first define what a transaction is and then place transaction API calls in the application based on the transactions that you define. For example, if you're writing a financial application, you might define a transaction to be the piece of code that queries an account balance. Another transaction might be the code that updates a bank account.

However, what if the bank account update requires a query of the account balance? Do you define two separate transactions or one transaction that includes both operations? The answer is whatever you define it to be. It depends on how much granularity you want in the measurement of the operations in your application. If you query the account balance for other reasons (in addition to updating the bank account), you might want to identify the query as a separate transaction. You decide. The transaction APIs will measure the transactions that you define.

The transactions you choose to define will obviously be different from application to application. For example, your retail application might retrieve daily sales data from its stores every night. In this case, you could define a "retrieve sales data" transaction in which the retrieval from each store is tracked and measured as a separate transaction. Likewise, an application that obtains price quotes from several suppliers could track and measure the time and resources used to obtain each price quote. The definition of a transaction is limited only by the way you execute work in your application.

Placing Transaction API Calls in the Application

After you define the server transactions you want to measure in your application, you place API calls at the beginning and end of each transaction in your code. The response time, CPU time, I/O activity, and seize/lock activity are calculated between each start and end transaction pair.

The Start Transaction API, which is named QYPE STRT (for OPM programs) or `qypeStartTransaction` (for ILE programs), is placed at the beginning of a transaction. The End Transaction API, which is named QYPEENDT or `qypeEndTransaction`, is placed at the end of a transaction. If the processing for a single transaction occurs in multiple threads or programs, you can use the Log Transaction API to keep track of a transaction as it moves from thread to thread or program to program. The Log Transaction API is named QYPELOGT or `qypeLogTransaction`. The details of the transaction APIs are described in the iSeries Information Center at publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?lang=en. Just click Programming|APIs|APIs by Category|Performance Management|PEX APIs.

Each API contains a parameter called Application Trace Data, which allows a programmer to capture any pertinent information that will help a performance analyst learn something important about the transaction. In the performance collection example shown later in this article, we used the Application Trace Data parameter to capture the user ID of the user performing the transaction and the customer ID accessed by the transaction. This data can be valuable when analyzing the cause of a performance bottleneck.

For example, you might find that transactions executed by a specific user or for specific customers have poor performance. Capturing this data in the transaction may help you identify who or what is causing poor performance. Other valuable information to include in this parameter might be the name of the file or the record number that is accessed by the transaction.

Code Example

To ease PEX API deployment and simplify the API call interface, we have wrapped up the PEX APIs in service program PEX001. (All of the code accompanying this article is available at iSeriesNetwork.com/code.) In the service program, all the PEX APIs are prototyped and made available through subprocedures that you can easily call from any program(s) you want to run a transaction performance analysis against.

The service program also offers a transaction ID counter facility. The transaction ID is a sequential number that uniquely identifies each set of Start Transaction, Log Transaction, and End Transaction API calls within a given PEX session. The transaction ID counter function involves three procedures:

- *Initialize Transaction ID*, which takes a counter ID (identifies one counter sequence) and returns a pointer to the counter storage
- *Get Transaction ID*, which takes the pointer and returns the next transaction ID (using a couple of MI functions to ensure that only one process at a time is allowed to reference the counter storage process)
- *Terminate Transaction ID*, which releases the resources used as counter storage (at this point, the transaction ID must be reinitialized to be used again; to keep transaction IDs unique throughout a PEX session, do not terminate the transaction ID until you no longer need to collect any more transaction data for that session)

See the PEX001 source header for instructions on how to create the service program.

[Figure 1](#) and [Figure 2](#) show the code snippets that should go into the program(s) you want to include in your analysis. The application trace data and performance counter parameters are provided only as an example of how these parameters could be used.

The prototypes for the PEX001 service program are stored in the copybook ([Figure 1](#)). Putting the prototypes in a copybook lets you quickly include them in the programs where they are needed, without having to code them over and over again.

[Figure 2](#) shows an example of how you would use the PEX API functions to collect performance data. To make the PEX functions in the PEX001 service program available to your program, specify the binding directory and copybook instructions (at A). The transaction counter, which is needed to generate unique transaction IDs for the performance data collection, is initialized at B.

At C, the beginning of the transaction to be measured is recorded by calling the StrTrnPex subprocedure, and at the point where the end of transaction is encountered, the EndTrnPex subprocedure is called (at D). The PEX facility will now have registered all the relevant performance data related to that specific transaction. When the performance data collection is complete, this information can be extracted into a number of data files and used to generate performance analysis reports, as we will show in a moment.

Collecting Transaction Performance Data

Once the transaction API calls are placed in the application, you can collect the transaction performance metrics while the application is running by following these steps:

1. Use the ADDPEXDFN (Add PEX Definition) command to define the transaction data that should be collected by PEX.
2. Use the STRPEX command to start PEX, which enables the collection of the transaction data that was defined in Step 1.
3. Run your application.
4. Use the ENDPEX command to end the collection of transaction metrics by PEX, and write the transaction data to the PEX database.

As we mentioned earlier, you can run PEX in many different modes at different levels of detail. As a result, there are many ways to use the ADDPEXDFN command. Because our purpose is to collect application transaction data, the ADDPEXDFN command should be called as follows:

```
ADDPEXDFN DFN (APPTXN)
TYPE (*TRACE) JOB ( (*ALL) )
TRCTYPE (*SLTEVT) SLTEVT (*YES)
OSEVT ( (*USRTNS) )
```

The DFN parameter contains the name of the definition that will be used on the STRPEX command to identify which type of data should be collected. The OSEVT parameter indicates that user (application) transaction data (*USRTNS) will be collected. The JOB parameter with a value of *ALL indicates that transaction data will be collected for all jobs running on the system. If your application contains a predefined set of jobs, you can specify individual job names instead of *ALL.

Next, start the collection of performance data by calling the STRPEX command as follows:

```
STRPEX SSNID (APPTXN1)
OPTION (*NEW) DFN (APPTXN)
```

The SSNID parameter is a value that uniquely identifies this session of data collection. The DFN parameter is the same as the DFN parameter on the ADDPEXDFN command.

After PEX is started, transaction data will be collected for any application that contains the PEX API calls. When the PEX session is ended, transaction data is stored in four database files named QAYPEMIUSR, QAYPETASKI, QAYPETIDX, and QAYPERUNI.

The transaction performance data is not stored in the database files until the ENDPEX command is run as follows:

```
ENDPEX SSNID (APPTXN1) OPTION (*END)
DTAOPT (*LIB) DTALIB (TXNLIB)
DTAMBR (*SSNID) RPLDTA (*YES)
TEXT ('My Transaction Data')
```

In this example, the transaction data is stored in library TXNLIB, specified by the DTALIB parameter. The DTAMBR parameter specifies the database file member name for the data. In this

case, the member name is the same as the value on the SSNID parameter, which is APPTXN1. The SSNID parameter must be the same as the SSNID parameter entered on the STRPEX command.

Note: When you run PEX for the first time, it will create all the database files for every mode of PEX that could possibly be used. This results in around 50 files being created, even though only four of the files are used for application transaction events.

Although you could technically collect transaction data continuously, a common way to use this facility is to collect the data for one or two hours during a peak period of the day. In this case, you will need to delay the call of the ENDPEX command until the application has run for the period of time that you determine in advance.

After transaction data has been collected, you can process and analyze the data with SQL queries. As we mentioned earlier, PEX produces four database files that are important when you're analyzing transaction data, but most of the performance information is found in the following two files:

- **QAYPEMIUSR.** This file contains the raw transaction event data. [Figure 3](#) and [Figure 4](#) show the format of this data.
- **QAYPETIDX.** This file contains the timestamp, CPU number, and TDE (task ID) number for each event or transaction that was produced during this data collection.

Files QAYPEMIUSR and QAYPETIDX are joined by the record number field, which is named QRECN in both files. The files must be joined in this way when you're performing the SQL query needed to analyze the transaction data.

Defining SQL Aliases

Before you run the query, define an SQL ALIAS for each member you need to access, because SQL does not directly support multimember files. Another technique is to use the OVRDBF CL command. Here are the SQL ALIAS definitions needed for this analysis:

```
CREATE ALIAS MIUSR FOR
  TXNLIB/QAYPEMIUSR (APPTXN1)

CREATE ALIAS TIDX FOR
  TXNLIB/QAYPETIDX (APPTXN1)
```

Note: You need to have the library name used here (TXNLIB) in your library list for the query shown in the next section.

Running the Query

Now you're ready to run the query. From your SQL session, build your query by including the fields you want to examine. In [Figure 5](#), a subset of the fields in the raw transaction file QAYPEMIUSR is included along with the timestamp from file QAYPETIDX. The timestamp is used to determine the overall response time of a transaction. The query results are ordered by the timestamp and transaction ID.

You might want to query on more fields than appear in [Figure 5](#), such as CPU time, the lock counters, and the thread information if a job contains multiple threads. However, in this example, we will focus on response time, database lock activity, and the application-specific trace data, which includes the user ID that performed the transaction and the customer ID. [Figure 6](#) shows the results of a query run after data was captured for two applications that contain the PEX transaction APIs.

The first application has an application ID of FinanceApp. This application analyzes the creditworthiness of a customer by investigating payment history, credit limit, and other financial criteria. The second application has an application ID of OrderEntryApp. This application checks current inventory and fills the order for a customer if the inventory is available.

In [Figure 6](#), transactions 9896 and 9900 are the longest running transactions. Both transactions take approximately 0.28 seconds, while most other transactions shown here take 0.10 seconds or less. You can determine this by looking at the timestamps for the start and end of each transaction.

The third record in the file represents the start time for transaction 9896 (because the TXNTYPE field is 1, which means the Start Transaction API was called). The fourth record in the file represents the end time for transaction 9896 (because the TXNTYPE field is 2, which means the End Transaction API was called). Computing the difference between the timestamp at the start of the transaction and the timestamp at the end of the transaction will give you the total response time for this transaction.

So what could have caused transactions 9896 and 9900 to take much longer than the other transactions? Notice that the transactions performed 677 and 674 asynchronous database reads. Again, this is computed by taking the difference of the counters at the start and the end of the transaction. Notice that all other transactions performed fewer than 200 reads.

You can see additional database counters selected on the query statement by using F20 to scroll to the right ([Figure 7](#)). Notice that the transactions identified above also performed the most asynchronous database writes and several synchronous database writes, which typically means that the data needed for the transaction was not in main memory, so the system needed to read database records into memory from disk. All of these discoveries, identified by querying the PEX transaction data, help explain why certain transactions took longer than others.

In this case, the application developer will want to investigate the application that performed the long-running transactions to determine whether improvements can be made to the application itself or to the data being accessed by these transactions. Notice that column 1 in [Figure 6](#) identifies that both transactions were run by the application identified as FinanceApp.

One Last Tip

Querying each transaction might be more detail than you need in some cases. You might first want to start by obtaining information about average transaction response time and throughput. Then, if you discover the response time or throughput rate is poor, you can use the techniques shown earlier to investigate individual transaction performance.

To obtain average response time and throughput statistics, you need to run Collection Services and query the file QAPMUSRTNS. You can configure Collection Services to collect performance data at regular intervals between 15 seconds and one hour.

The format of file QAPMUSRTNS is described in the iSeries Information Center at Systems management|Performance|Applications for Performance Management|Performance database files|Data files containing time interval data|QAPMUSRTNS. You can also find more information about Collection Services at Systems management|Performance|Applications for Performance Management|Collection Services.

Improve Application Efficiency

Measuring the transaction performance for non-5250 applications can be a complex task, but the new PEX Transaction APIs in i5/OS make this task easier for applications running on an iSeries server. If you use the APIs in your applications, you will be able to capture transaction throughput, response time, CPU consumption, and other important performance statistics. Having this data at your fingertips lets you do a better job of planning capacity and identifying performance bottlenecks, which can result in more efficient applications and satisfied users.

Keith Zblewski is a performance analyst and software architect in the POWER Systems Performance organization at IBM in Rochester, Minnesota. He joined IBM in 1987 and has been involved with various aspects of iSeries and eServer performance since 1997. Keith currently leads the development of sizing tools for i5/OS, AIX, and Linux and provides technical guidance to IBM and Business Partner sales and support teams on iSeries performance and capacity planning.

Carsten Flensburg has been an iSeries programmer since 1992. He currently works as an iSeries programming team leader for a European vacation rental company called Novasol, which is a part of the U.S.-based Cendant Corporation. Carsten lives in Copenhagen, Denmark, with his wife, Dorte, and his two children, Julian and Emilie.

Source URL: <http://iprodeveloper.com/rpg-programming/using-pex-apis-trace-application-specific-transaction-performance>