

PowerServer Performance Tuning Guide

Appeon® PowerServer® 2020
FOR WINDOWS & UNIX & LINUX

DOCUMENT ID: ADC10089-01-06780-01

LAST REVISED: March 25, 2020

Copyright © 2020 Appeon. All rights reserved.

This publication pertains to Appeon software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Appeon Inc.

Appeon, the Appeon logo, Appeon PowerBuilder, Appeon PowerServer, PowerServer, PowerServer Toolkit, AEM, and PowerServer Web Component are trademarks of Appeon Inc.

SAP, Sybase, Adaptive Server Anywhere, SQL Anywhere, Adaptive Server Enterprise, iAnywhere, Sybase Central, and Sybase jConnect for JDBC are trademarks or registered trademarks of SAP and SAP affiliate company.

Java and JDBC are trademarks or registered trademarks of Sun Microsystems, Inc.

All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Appeon Inc., 1/F, Shell Industrial Building, 12 Lee Chung Street, Chai Wan District, Hong Kong.

Contents

1	Appeon Performance	1
1.1	Impacts to Appeon performance	3
1.1.1	Impact of the Internet and slow networks	3
1.1.2	Impact of #heavy# client-side logic	4
1.1.3	Impact of large data transmission	5
1.2	Expected performance level	6
1.3	Automatic performance boosting	6
2	Performance-Related Settings	8
2.1	Overview	8
2.2	PowerServer Toolkit performance settings	8
2.3	AEM performance settings	8
2.3.1	Timeout settings	8
2.3.2	DataWindow data caching	10
2.3.3	Multi-thread download settings	10
2.3.4	Custom Libraries download settings	10
2.3.5	Log file settings	10
2.4	Internet Explorer performance settings	10
2.5	Web and application server performance settings	11
2.5.1	Microsoft IIS server	11
2.5.1.1	Recommendations for avoiding common errors on IIS	11
2.5.1.2	Advanced thread settings	12
2.6	Database performance settings	14
2.6.1	Recommended database driver	14
2.6.2	Recommended database setting	15
3	Identifying Performance Bottlenecks	16
3.1	Overview	16
3.2	Analyzing log files	16
3.2.1	Analyzing Windows application log files	16
3.2.2	Analyzing PowerServer log files	16
3.2.3	Analyzing active transaction log	17
3.3	Identifying Performance Bottlenecks of Web Server and Application Server	18
3.4	Identifying Performance Bottlenecks of DB Server	18
3.4.1	Deadlock analysis	18
3.5	Identifying Performance Bottlenecks of PB application	19
3.5.1	Analyzing performance bottlenecks of PB application	19
4	Tuning: DB Server	20
4.1	Database	20
5	Tuning: Excessive Server Calls	21
5.1	Overview	21
5.2	Technique #1: partitioning transactions via stored procedures	21
5.3	Technique #2: partitioning non-visual logic via NVOs	23
5.4	Technique #3: eliminating recursive Embedded SQL	25
5.5	Technique #4: grouping multiple server calls with Appeon Labels	26
6	Tuning: Heavy Client	32
6.1	Overview	32

6.2	Technique #1: thin-out #heavy# Windows	32
6.3	Technique #2: thin-out #heavy# UI logic	32
6.3.1	Manipulating the UI in loops	32
6.3.2	Triggering events repeatedly	33
6.3.3	Performing single repetitive tasks	33
6.3.4	Initializing #heavy# tabs	34
6.3.5	Using ShareData or RowsCopy/RowsMove for data synchronization	34
6.3.6	Using computed fields	34
6.3.7	Using DataWindow expressions	34
6.3.8	Using complex filters	35
6.3.9	Using RowsFocusChanging/RowsFocusChanged events	35
6.4	Technique #3: offload #heavy# non-visual logic	35
7	Tuning: Large Data Transmissions	36
7.1	Overview	36
7.2	Technique #1: retrieving data incrementally	36
7.2.1	For Oracle database server	36
7.2.2	For all other database servers	37
7.3	Technique #2: minimizing excessive number of columns	37
8	Conclusion	38
	Index	39

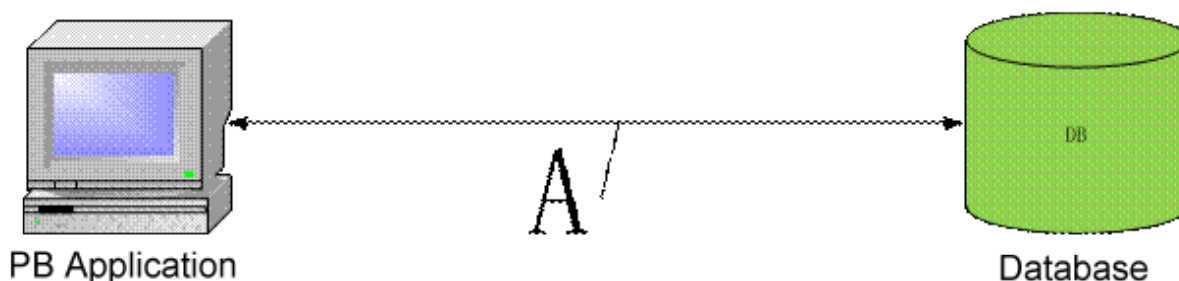
1 Apeon Performance

Before we officially introduce the methods to analyze and optimize the performance, we spare a few minutes to look at the possible impacts that cause performance problems to PowerServer-converted applications. This helps to learn optimization solutions better, or even helps to work out individual optimization solutions for each application.

Traditional PowerBuilder application architecture

The traditional PowerBuilder application is based on the client/server architecture, in which applications run on the client side and interact with the database with native drivers. The connection between the client and the database (marked as "A" in the following figure) is usually the local area network or high-speed enterprise network, therefore the connection can hardly impact the performance. In this architecture, the main performance impacts are the application and the database.

Figure 1.1: Traditional PB application architecture



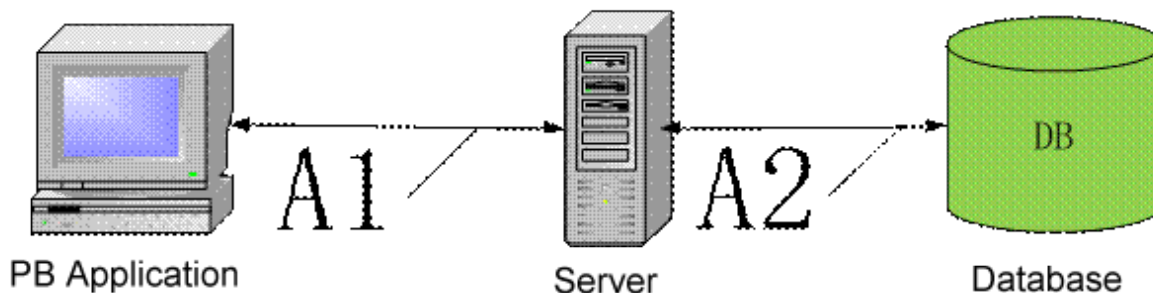
Thus, the performance of C/S PB applications = PB client performance + A + database processing performance.

Since the connection impact can be negligible, the performance of C/S PB applications = PB client performance + database processing performance.

PowerServer (Web/Mobile) application architecture

Compared with the traditional C/S architecture, the PowerServer application architecture has a server (that holds the core business logic and the data service) between the client and the database. The client does not directly interact with the database anymore, but sends all the requests and data to the server, and interacts with the database through the server.

Figure 1.2: PowerServer application architecture



In the architecture, the PowerServer application performance = Web/Mobile client performance + A1 + server performance + A2 + database processing performance.

A1 is the network connection between the client and the server, and is usually the wide area networks/internet. Therefore, this is a very large performance expense, especially when the client frequently interacts with the server.

A2 is the network connection between the server and the database, and is usually the local area networks or high-speed enterprise networks. The same as that in the C/S architecture, it can hardly impact the performance.

Therefore, the PowerServer application performance = Web/Mobile client performance + A1 + server performance + database processing performance.

PowerBuilder application impacts vs. PowerServer application impacts

According to the PowerBuilder application architecture and the PowerServer Web/mobile application architecture described above, we know that:

- PB application performance = PB client performance + database processing performance
- PowerServer application performance = Web/Mobile client performance + A1 + server performance + database processing performance

PowerServer applications are converted from PB applications and are consisted of two parts after migration: the Web/Mobile client and the server. Therefore, the PowerServer application performance is dependant on the PB application performance.

Two hypotheses as follows:

1. The PB application does not have performance problems; but the converted PowerServer application has.

In this case, the PB client and the database performance expense are negligible. Since the performance of PowerServer Web/Mobile client and the server are dependent on the PB client performance, the performance problem may be caused by A1, the connection network.

And the possible reasons are:

- The networks connection is slow or unstable;
- The data package is too large or the SQL syntaxes are not efficient that result in long communication time in a single communication;
- The same functionality frequently communicates with the server that results in repeated connection performance (A1) expense, etc.

In the case, developers should: 1) first consider to reduce the communication times between the client and the server so to reduce the connection performance expense; 2) secondly, consider to optimize the efficiency of each communication, for example, by retrieving only the necessary data and using the optimal relational calculus in the SQL syntaxes, etc.

Usually, if the PB application does not have performance problems, the PowerServer application converted from it will not have problems either. But under certain circumstances, there still can be performance problems in the converted PowerServer application. This is because the PB application will finally run using the machine code

or the quasi machine code, which is very efficient at code levels; but the converted PowerServer application executes the JavaScript, an interpreted code, and will interpret at runtime. Therefore, the efficiency is lower compared with the machine code, especially when there is

- Functionalities that use multiple character string manipulation;
- Lots of loops;
- Frequently-distributing memories or large memories, for example, large objects, lots of tabpages in the tab control, etc.

2. Both the PB application and the converted PowerServer application have performance problems:

If the PB application has performance problems, the converted PowerServer applications will definitely have performance problems as well.

In this case, developers should:

1) first consider to optimize the performance of the PB application and the database by using all kinds of available system tools. For example, developers can use the transaction track analyser provided by the database provider to analyze and optimize the database performance. Usually, popular database providers provide performance analysis and optimization tools with their databases, developers can use these provided tools to optimize the databases.

2) secondly, after you make sure that the PB application does not have performance problems, use the hypothesis 1 to analyze and the converted PowerServer application.

In details, the impacts to the converted PowerServer application are as follows:

- [Impact of the Internet and slow networks](#);
- [Impact of "heavy" client-side logic](#);
- [Impact of large data transmission](#).

Refer to the [Impacts to Appeon performance](#) for more information.

1.1 Impacts to Appeon performance

1.1.1 Impact of the Internet and slow networks

Although Appeon pushes the envelope to deliver unparalleled performance from standard Web technologies (e.g. XML, JavaScript, HTML, Java or C#), which are typically significantly slower than PowerBuilder, slow and latent network connections rob performance from even the best applications!

Network chatter and network-intensive code really highlight the weakness of a poor network connection. Any code that results in a HTTP request (i.e. server call) when executed multiple times sequentially has potential to create network chatter. There are mainly two categories of code that result in server calls - data access related and remote method invocations. Here are several common examples so you can familiarize yourself:

- Embedded SQL (Select, Insert, Delete, Update, Cursor) including Dynamic SQL;
- Invoking stored procedures or database functions;
- DataWindow/DataStore Functions (Retrieve, Update, ReselectRow, GetFullState, SetFullState, GetChanges, SetChanges);
- DataWindow/DataStore Events (SQLPreview);
- Invoking a method of a server-side object, such as a PowerBuilder NVO, Java EJB, or .NET Component; or
- Invoking a Web Service.

Each of the above statements will generate one call to the server utilizing HTTP, with exception of SQLPreview event that will generate one call for each line of code handled by the event. If any of the above statements are contained in a loop or recursive function, well depending on the number of loops, even though its just one statement it would be executed multiple times generating multiple server calls. Needless to say, loops and recursive functions are some of the most dangerous from a performance perspective.

The reason it is important to minimize server calls is because it can take 100 or even 1,000 times longer to transmit one packet of data over the Internet compared to a LAN. Imagine an event handler is triggered, for example handling an "onClick" event, whose execution will result in 80 synchronous server calls over a LAN with latency of 2 milliseconds (ms). In such scenario the slow-down attributed to network latency would be 0.16 seconds (80 x 2 ms). Now imagine this same event handler running over a WAN with latency of 300 ms. The slow-down attributed to the network latency would be a whopping 24 seconds (80 x 300 ms)! And depending on the amount of data transmitted there could be additional slow-down due the bandwidth bottlenecks.

It is imperative for the developer to be conscious that PowerBuilder applications deployed to the Web may not be running in a LAN environment, and as such there will be some degree of performance degradation. How much depends on how the code is written, but in most cases the performance degradation still falls within acceptable limits without much performance optimization.

Should you find that certain operations in your application are unacceptably slow, the good news is there are numerous things that you can do as PowerBuilder developers to ensure your PowerBuilder applications perform well in a WAN environment (e.g. Internet) or on slower networks. At a high-level, your code needs to be written such that the server calls and other performance intensive code is minimized or relocated to the middle-tier or back-end. This will be covered in more detail in the following chapters. Some changes are actually quite simple while others may require increased effort. Nonetheless, in all cases optimizing the performance of your applications in PowerBuilder is just a fraction of the effort to work with typical low-productivity Web tools such as VisualStudio.NET and Eclipse.

1.1.2 Impact of #heavy# client-side logic

Most PowerBuilder applications are developed utilizing a 2-tier architecture. In other words, all the PowerScript and embedded SQLs are coded in the Visual objects, for example Window, CommandButton, etc. In contrast, a 3-tier architecture would encapsulate all

non-visual logic in PowerBuilder NVOs (Non-Visual Objects). The reality is even if your application utilizes NVOs, chances are it is not a pure 3-tier application if PowerBuilder NVOs are not exclusively utilized to encapsulate all non-visual logic. But don't rush to partition your application just yet!

Most applications developed as a 2-tier architecture perform great in Apeon. In fact, there are many situations that a 2-tier application when deployed by PowerServer will actually perform faster than a 3-tier application. The reason is if a PowerBuilder NVO is deployed to the middle-tier or application server, time must be spent to call the server and get the results back to the client. Of course, your non-visual logic running on an application server will run faster than at the Web browser. The key question is how much performance do you gain by running a particular block of code on the application server vs. how much performance do you lose due to the server calls.

As a rule of thumb, it is recommended to partition your non-visual logic to the middle-tier only when the particular block of code runs unacceptably slow at the Web browser. In such cases, it is likely that the application performance will benefit, and as such, it is worthwhile to invest the time to partition such logic. However, if the non-visual logic is only slightly sluggish, it may be possible to optimize the code without having to partition it to the application server.

1.1.3 Impact of large data transmission

When you first open a Window there are two types of files downloaded. The first type is the HTML and JavaScript files ("Web files") that contain the UI and UI logic of the application Window. The second type is data files that contain the result set, for example for a DataWindow retrieve. The time to download these files is affected by two factors: 1) the network connection and 2) the size of the files to be downloaded.

The Web files do not impact performance because of their small size and the enhanced ability of the browser to "cache" them. The Web files for a given PowerBuilder Window are typically between 25-75 KB. Because these Web files are static in nature, once a given application Window has been opened, the Web files will be cached on the Client computer. As such, once these Web files are cached, their impact on performance is essentially non-existent.

Under most circumstances, these Web files are not re-downloaded when the Window is reopened. The only exceptions are if 1) the temporary Internet files folder has been emptied or 2) the application has been updated and redeployed to the server. If the latter has happened, Web files for only those Windows that have been modified will be automatically downloaded from the Web server.

Only the data files containing the result sets may or may not be cached (depending on whether you have enabled DataWindow caching). A result set of 50 records would typically result in a 12 KB data file. Every 5 records would typically add another 1.2 KB to the data file size. So, for example, a 500-record result set would typically correspond to a 120 KB data file. If DataWindow caching is not enabled, the data files corresponding with such DataWindows will be downloaded from the server each time a DataWindow retrieve is invoked.

The good news is that Apeon has built-in 10X data compression for DataWindow result set to essentially eliminate the time spent downloading these data files. The same 500-record

result set that would normally correspond to a 120 KB data file would only result in the download of a 12 KB data file from the server. This compression feature makes even the largest of result sets quick to transfer.

In conclusion, due to Web file caching feature of the Web browser, Appeon's built-in DataWindow caching technology and 10X data compression technology, generally speaking neither the Web files nor data files should have any noticeable impact on the performance of your Web application.

1.2 Expected performance level

When comparing a PowerBuilder application to the performance of a Web application deployed by PowerServer to a LAN environment, generally speaking the performance of the two will be quite similar. In some cases (for certain operations) Appeon may actually be even faster than PowerBuilder.

The reason is that Appeon has been tuned for nearly a decade to offer the best performance possible for real-life PowerBuilder applications:

- Large PowerBuilder applications up to 600MB (of PBLs) including several thousand DataWindows and thousands of Windows.
- Complex screens containing as much as 80 DataWindows in a single Window
- Dynamically created objects (DataWindows, UserObjects, etc.)
- PFC and other high-overhead frameworks similar to PFC.

1.3 Automatic performance boosting

Appeon has a number of features built into its infrastructure/framework to instantly or automatically boost the performance of PowerBuilder applications when deployed to the Web. Many of these features are always on and transparently working in the background to boost performance. Other features are user-selectable and must be configured. The following table is a list of these features and the configuration of these features is covered in [Chapter 2, Performance-Related Settings](#).

Table 1.1: Appeon Performance Boosting Features

Performance Feature	Description	Location
Just-in-Time Downloading	As the application is run and various windows are opened, only the Web files required for that particular window are downloaded at that point in time. Once the Web files are downloaded, they are cached in the Internet Explorer temporary files folder and are not downloaded again.	Appeon Infrastructure
10X Web File Compression	All JavaScript files are compressed by as much as 10X, then the compressed version of the file is downloaded over HTTP to the Web browser.	PowerServer Toolkit

Performance Feature	Description	Location
10X Data File Compression	For each DataWindow or DataStore retrieval, the result set is first retrieved by the application server, automatically compressed by 10 times in most cases, and then downloaded over HTTP to the Web browser. Utilizing AJAX technology, only the DataWindow or DataStore is refreshed and the rest of the screen remains intact.	Appeon Infrastructure
DataWindow Data Caching	For each DataWindow or DataStore, the developer has the option of enabling caching of the result set. Appeon enables caching at the application server, Web server, and Web browser so every tier of the Web architecture is benefiting from the best performance and scalability possible.	AEM
Merge files	Merges multiple JavaScript files into a single file to reduce the number of HTTP requests and corresponding overhead.	PowerServer Toolkit
Multi-thread Downloading	Downloads are multi-threaded to boost the application runtime performance.	AEM
Custom Libraries Downloading	Any custom libraries can be automatically downloaded and installed with your Web application, or if the libraries are very large in size, you can disable this feature and distribute the libraries some other fashion.	AEM
Database Connection Pooling	By deploying to a true n-tier Web environment with Appeon, you can take advantage of Database Connection Pooling, a feature of most application servers. Connection Pooling does exactly that, it establishes a pool of connections to your database, which is shared among your clients. So rather than each client having its own dedicated connection to the database, a fewer number of connections can be rotated among all the users. For large deployments with thousands of clients this can boost database scalability noticeably.	Appeon Infrastructure

2 Performance-Related Settings

2.1 Overview

Performance settings need to be configured in PowerServer Toolkit, AEM, Web browser and your application server to ensure good performance in a production environment. If you identify any performance bottlenecks, it is strongly recommended that you first ensure all performance-related settings are correctly configured. Only if the performance issues still persist after the performance settings are configured correctly, then it is recommended to consider optimizing your PowerBuilder code.

2.2 PowerServer Toolkit performance settings

The following table lists the Apeon performance settings that the developer can configure (in the application profile) of PowerServer Toolkit to boost performance. To make a performance setting effective for an application, enable the option in the application profile and then perform a "Full Deployment" of the application.

Table 2.1: Performance settings in PowerServer Toolkit

Performance Feature	Description	User-Selectable
10X Web File Compression	Compresses files when they are transferred over the network.	User must enable this feature for it to become effective.
Merge files	Merges the small files during the application deployment. The small files will be downloaded to the client in one file package at one call, instead of being downloaded one by one at separate calls.	User must enable this feature for it to become effective.
Download ActiveX files in a single thread	Downloads the two ActiveX files in the same thread at runtime. Using the same thread to download the two ActiveX files can speed up the download under particular network conditions.	User must enable this feature for it to become effective.

2.3 AEM performance settings

2.3.1 Timeout settings

By setting proper values for Timeout Detection Interval and Timeout Settings in AEM, PowerServer can release the timeout and invalid database connections in time, thus can avoid database deadlock or malfunctions, so that the concurrent processing ability and the running stability of PowerServer for .NET can be greatly enhanced.

2.3.1.1 Session Timeout Detection Interval

The Session Timeout Detection Interval setting is to specify the interval (in seconds) to check whether sessions have timed out.

It is recommended that the value is smaller than the Session Timeout value. The default value is 30 seconds. The value recommended by the system is based on your Session Timeout values, if the Session Timeout values are different from applications, the system will multiply the smallest value with 0.15 to get the recommended value.

2.3.1.2 Transaction Timeout Detection Interval

The Transaction Timeout Detection Interval setting is to specify the interval (in seconds) to check whether transactions have timed out.

It is recommended that the value is smaller than the Transaction Timeout value. The default value is 30 seconds. The value recommended by the system is based on your Transaction Timeout values, if the Transaction Timeout values are different from applications, the system will multiply the smallest value with 0.15 to get the recommended value.

2.3.1.3 Session timeout

Session timeout ends the user session and rolls back all database updates since the last commit for a user session. The default value is 3600 seconds. Session Timeout should be larger than the Transaction Timeout setting. Generally speaking, Session Timeout should not be smaller than 3600 seconds.

2.3.1.4 Transaction timeout

Transaction timeout rolls back all database updates since the last commit in a transaction. The default value is 120 seconds. Transaction Timeout should be less than Session Timeout.

If transaction timeout in the application database is set to 1800 seconds, then Transaction Timeout in AEM should be set to 1810 or larger.

If transaction timeout in the application database is not set, then Transaction Timeout in AEM should be set to a number greater than the maximum time needed to execute regular database operations for the Web application, suppose the most time-consuming table query operation takes 3600 seconds to complete, then Transaction Timeout should be set to 3610 or larger.

2.3.1.5 Rollback Completion time

Specifies the maximum time (in seconds) to complete a rollback of a transaction. If the time to roll back transaction exceeds the specified value, the transaction rollback will be terminated. The default value is 3 seconds.

It is recommended that it is set to a number between 3 seconds and Transaction Timeout *0.1.

2.3.1.6 Maximum Rollback Retries

The maximum times to retry a rollback of a transaction. PowerServer executes a rollback if a transaction times out. If the rollback fails, PowerServer will keep retrying until this maximum value is reached. Setting this value to 0 will disable this feature. The default value is 3.

It is recommended that it is set to a number smaller than 10.

2.3.2 DataWindow data caching

AEM provides a DataWindow data cache mechanism for caching the frequently used DataWindow data. It is recommended that you enable the cache for the DataWindow objects whose data is relatively static. Any DataWindow objects whose data is fairly dynamic should remain unchecked, otherwise you will experience overhead from the caching mechanism without the true benefits of the caching.

NOTE: DataWindow Data Cache will not be effective until you fulfill all the configuration requirements that are detailed in Section 4.4.8.3, “DataWindow Data Cache” in *PowerServer Configuration Guide for .NET* or in *PowerServer Configuration Guide for J2EE*. Also, this feature is only supported on Windows servers.

2.3.3 Multi-thread download settings

Multi-thread downloads boost the application runtime performance. However, if there are many threads competing for the processing power of Web server, it may slow down the performance of Web server. Therefore, you should not specify an unnecessarily large number of threads in AEM. It may take some trial and error to fine-tune the performance.

2.3.4 Custom Libraries download settings

If your application utilizes a customer library (e.g. DLL, OCX, etc.), you can specify in AEM how the custom library should be downloaded to the client:

- In most situations you should set the install mode to "Install automatically without asking user" or "Confirm with user, then install automatically". With these options, the custom libraries will be automatically downloaded and seamlessly installed to the Web browser.
- However, if the file size of the custom libraries is extremely large (e.g. tens of megabytes), you should set the install mode to "Install manually (no automatic installation)". With this option, PowerServer does not automatically download and install the custom libraries. As such, you must distribute the custom libraries to users and your users need to install it manually.

For more details about the custom libraries download settings, refer to Section 4.4.4.3, “DLL/OCX Files” in *PowerServer Configuration Guide for .NET* or in *PowerServer Configuration Guide for J2EE*.

2.3.5 Log file settings

Once your application is fully tested and ready to move to a production environment, it is recommended to disable the AEM log functionality. Writing log files incurs disk activity, which can impact performance. Generally, the impact is small but nonetheless it will not hurt to disable this.

2.4 Internet Explorer performance settings

For optimal performance, it is recommended that the Web file caching functionality of Internet Explorer be fully utilized. This will significantly reduce the time required to load and start an application following the initial load. The configuration outlined below will ensure

that you realize the best performance while safeguarding your application from becoming "stale".

Step 1: Open Internet Explorer and select **Tools > Internet Options**. Verify that the **Empty Temporary Internet Files folder when browser is closed** option is not checked under the **Security** section of the **Advanced** tab of Internet Options.

Step 2: Click the **Settings** button under the **General** tab to configure the Temporary Internet Files settings.

Step 3: Select the **Automatically** radio button and verify that the Amount of disk space to use scroll box is set to a reasonable number, such as 200 MB or more.

Now the browser is set to automatically check for newer versions of the Web application.

2.5 Web and application server performance settings

2.5.1 Microsoft IIS server

Tune the IIS server will help you to avoid some common errors and gain a better performance of Web applications. This section highlights the key performance settings of IIS. You can refer to the IIS Operation Guide for details on how to extensively tune IIS server.

2.5.1.1 Recommendations for avoiding common errors on IIS

IIS should be tuned to avoid the contention, poor performance, and deadlocks for the Web applications. Following are some samples of commons error that you may find in Application log, System log or even log in the Web browser.

- Event Type: Error

Event Source: ASP.NET

Event ID: 1003

Description: aspnet_wp.exe (PID: <xxx>) was recycled because it was suspected to be in a deadlocked state. It did not send any responses for pending requests in the last 180 seconds.

- Event Type: Warning

Event Source: W3SVC-WP

Event ID: 2262

Description: ISAPI 'C:\Windows\Microsoft.net\Framework\v.2.0.50727\aspnet_isapi.dll' reported itself as unhealthy for the following reason: 'Deadlock detected'.

- Event Type: Warning

Event Source: W3SVC

Event ID: 1013

Description: A process serving application pool 'DefaultAppPool' exceeded time limits during shut down. The process id was '<xxxx>'.

- You may receive the exception error message: "System.InvalidOperationException: There were not enough free threads in the ThreadPool object to complete the operation."
- In the browser, you may also receive the exception error message: "Request timed out".

To avoid such problems, you can configure ASP.NET according to the suggestions below so to best fit your situation and make the Web application perform better. Details on how to configure the ASP.NET can be found at [Advanced thread settings](#).

1. Limit the number of .NET requests that can execute at the same time to approximately 12 per CPU.
2. Permit Web service callbacks to freely use threads in the ThreadPool.
3. Select an appropriate value for the [maxconnections](#) parameter. Base your selection on the number of IP addresses and AppDomains that are used.

2.5.1.2 Advanced thread settings

The following settings are the most-PowerServer-related settings in ASP.NET.

When you call a PowerServer Web application from IIS, you may experience contention, poor performance, and deadlocks. Clients may report that requests stop responding (or "hang") or take a very long time to execute.

This problem might occur because ASP.NET limits the number of worker threads and completion port threads that a call can use to execute requests. If there are not sufficient threads available, the request is queued until sufficient threads are free to make the request. Therefore, .Net will not execute more than following number of requests at the same time:

$(\text{maxWorkerThreads} * \text{number of CPUs}) - \text{minFreeThreads}$

Note: The minFreeThreads parameter and the minLocalRequestFreeThreads parameter are not implicitly multiplied by the number of CPUs.

2.5.1.2.1 maxWorkerThreads and maxIoThreads

.NET uses the following two configuration settings to limit the maximum number of worker threads and completion threads that are used:

```
<processModel maxWorkerThreads="20" maxIoThreads="20" />
```

The maxWorkerThreads parameter and the maxIoThreads parameter are implicitly multiplied by the number of CPUs, the default value of these two parameters are both 20. If for some reason your application is slow, perhaps waiting for external resources, you could try to increase the number of threads to a value less than 100. For example, if you have two processors, the maximum number of worker threads is the following: $2 * \text{maxWorkerThreads}$

2.5.1.2.2 minWorkerThreads

The setting determines how many worker threads may be made available immediately to service a remote request. By default, the minWorkerThreads parameter is not present in either the Web.config file or the Machine.config file at C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG. You need to manually add the following line to make the setting work.


```
<processModel minWorkerThreads="1" />
```

Threads that are controlled by this setting can be created at a much faster rate than worker threads that are created in other ways. The default value for the `minWorkerThreads` parameter is 1. The setting is recommended to set in the following way.

$$\text{minWorkerThreads} = \text{maxWorkerThreads} / 2$$

Note: This setting is implicitly multiplied by the number of CPUs.

2.5.1.2.3 minFreeThreads and minLocalRequestFreeThreads

The two settings determine how many worker threads and completion port threads must be available to start a remote request or a local request:

```
<httpRuntime minFreeThreads="8" minLocalRequestFreeThreads="8" />
```

The default value is 8. If there are not sufficient threads available, the request is queued until sufficient threads are free to make the request. Therefore, .NET will not execute more than the following number of requests at the same time:

$$(\text{maxWorkerThreads} * \text{number of CPUs}) - \text{minFreeThreads}$$

Note: The `minFreeThreads` parameter and the `minLocalRequestFreeThreads` parameter are not implicitly multiplied by the number of CPUs.

2.5.1.2.4 maxconnection

The `maxconnection` parameter determines how many connections can be made to a specific IP address. The parameter appears as follows:

```
<connectionManagement>
  <add address="*" maxconnection="2" />
  <add address="65.53.32.230" maxconnection="12" />
</connectionManagement>
```

The `maxconnection` parameter setting applies to the `AppDomain` level. By default, because this setting applies to the `AppDomain` level, you can create a maximum of two connections to a specific IP address from each `AppDomain` in your process.

2.5.1.2.5 execution Timeout

The setting limits the request execution time:

```
<httpRuntime executionTimeout="90" />
```

The default is 110 seconds.

Note: If you increase the value of the `executionTimeout` parameter, you may also have to modify the `processModel responseDeadlockInterval` parameter setting.

2.5.1.2.6 Recommended thread settings

For most applications, you can use and apply the recommended changes in the `Machine.config` file as below, which can be found at `C:\WINDOWS\Microsoft.NET\Framework\v2.0.*****\CONFIG`:

1. Set the values of the `maxWorkerThreads` parameter and the `maxIoThreads` parameter to 100.

2. Set the value of the maxconnection parameter to $12 * N$ (N is the number of CPUs that you have).
3. Set the values of the minFreeThreads parameter to $22 * N$ and the minLocalRequestFreeThreads parameter to $19 * N$.
4. Set the value of minWorkerThreads to 50. Remember, minWorkerThreads is not in the configuration file by default. You must add it.

If you have hyperthreading enabled, you must use the number of logical CPUs instead of the number of physical CPUs.

Note: If you have a server with one processor, when you use this configuration, you can execute a maximum of 78 .NET requests at the same time because $100 - 22 = 78$. Therefore, at least $22 * N$ worker threads and $22 * N$ completion port threads are available for other uses (such as for the Web service callbacks).

For example, if you have a server with four processors and hyperthreading enabled, then $n = 8$ ($= 2 * 4$). Based on these formulas, you would use the following values for the configuration settings that are mentioned in this section.

```
<system.web>
.....
<processModel maxWorkerThreads="100" maxIoThreads="100" minWorkerThreads="50" />
<httpRuntime minFreeThreads="176" minLocalRequestFreeThreads="152" />
.....
</system.web>
.....
<system.net>
.....
<connectionManagement>
  <add address="[ProvideIPHere]" maxconnection="96" />
</connectionManagement>
.....
</system.net>
```

Also, if you use this configuration, 12 connections are available per CPU per IP address for each AppDomain, and you can execute a maximum of 624 .NET requests at the same time because $8 * 100 - 176 = 624$.

2.6 Database performance settings

2.6.1 Recommended database driver

The following database drivers are recommended for Apeon .NET:

Table 2.2: Recommended database drivers

Database Type	Recommended Driver
SAP SQL Anywhere 8.0.2, 9.0, or 10.0.1	ODBC driver
SAP SQL Anywhere 11.0, 12.0, 16.0, or 17.0	Native driver
SAP ASE	Native driver
MS SQL Server	Native driver
Oracle	Native driver

Database Type	Recommended Driver
Informix	Native driver
IBM DB2	Native driver
MySQL	Native driver
PostgreSQL	Native driver

2.6.2 Recommended database setting

Setting appropriate values for the database parameters based on the actual needs can reduce the occurrence of database deadlock and block hence can improve the concurrency and stability of the Web application.

2.6.2.1 Command Timeout

Specify the timeout period for the commands. In the following table, it shows how to set Command Timeout to 120 seconds in different databases.

Table 2.3: Command timeout settings in different databases

Database Type	Parameter	Remark
SAP SQL Anywhere	BLOCKING_TIMEOUT	Execute the following command: SET OPTION Public.BLOCKING_TIMEOUT = 120000
ASE	lock wait period	Execute the following command: sp_configure "lock wait period", 120
SQL Server	-	Modify the Command Timeout parameter in Edit Data Source page of AEM. Command Timeout = 120
Oracle	IDLE_TIME	Execute the following command: ALTER PROFILE default LIMIT IDLE_TIME 2;
Informix	DEADLOCK_TIMEOUT	Modify the DEADLOCK_TIMEOUT parameter in ONCONFIG.std file. DEADLOCK_TIMEOUT = 120

3 Identifying Performance Bottlenecks

3.1 Overview

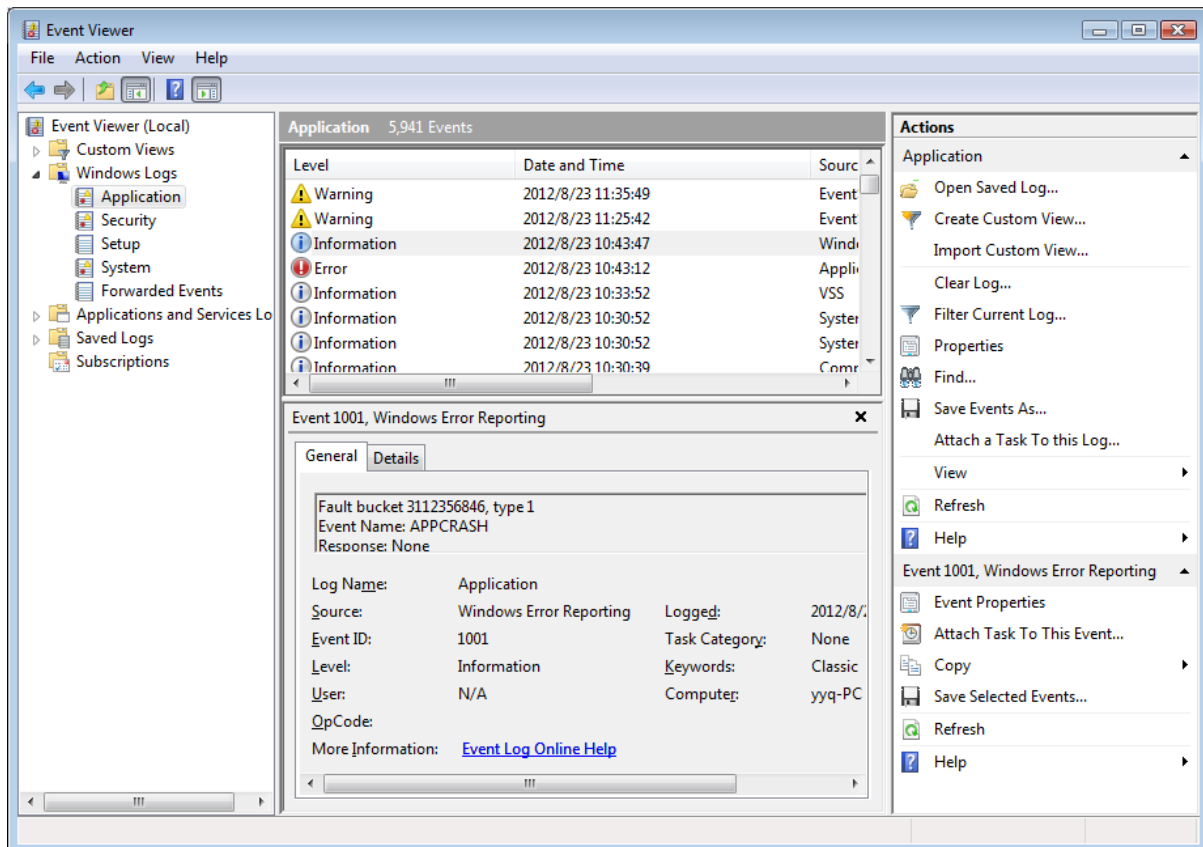
There are several methods to identify performance bottlenecks in your application. You can manually test your application or utilize Apeon's built-in performance reporting tool. Manually testing is the most time-consuming but also the most comprehensive and accurate. Nonetheless, the performance reporting tool, can help to identify most problematic Windows without a lot of work.

3.2 Analyzing log files

3.2.1 Analyzing Windows application log files

Check whether there are log files about the application server in the Event Viewer, then go to the application server web site to get the detailed solution to solve those Errors.

Figure 3.1: Event Viewer



3.2.2 Analyzing PowerServer log files

Step 1: Select Debug Mode as the Log Mode.

Figure 3.2: Log mode

Log Mode

Off

Standard mode (default)

Developer mode

Debug mode

Step 2: View the ApeonServer and ApeonError.log.

Figure 3.3: Viewing logs

[Welcome](#) > [Server](#) > [Logging](#)

Appeon Server Logs

Actions	Log	Size (KB)
<input type="button" value="View"/> <input type="button" value="Download"/> <input type="button" value="Clear"/>	Server Log	0.00
<input type="button" value="View"/> <input type="button" value="Download"/> <input type="button" value="Clear"/>	Error Log	0.00
<input type="button" value="View"/> <input type="button" value="Download"/> <input type="button" value="Clear"/>	Deployment Log	0.00

Read ApeonServer.log and find out operations which are time consuming and redundant.

3.2.3 Analyzing active transaction log

Step 1: Go to Active Transactions screen and select the intended Transaction ID.

Figure 3.4: Active Transactions

[Welcome](#) > [Server](#) > [Sessions](#) > [Active Transactions](#)

Active Transactions

Lists the active transactions for all Appeon Servers.

View active transactions for: all Appeon Servers

<input type="checkbox"/>	Kill Session	Transaction ID	Session ID	User name	IP Address	Application Name	Server ID	Duration (sec)
<input type="button" value="Rollback Checked Transactions"/> <input type="button" value="Refresh"/>								

Step 2: Go to Active Transaction Log, you can view more details about what this transaction acted on the database recently.

Figure 3.5: Active Transaction Log

Active Transaction Log				
Display the detail information of the transaction.				
Session ID	Transaction ID	Logged Time	Action	Details
2061427769	sqlca	2012-04-28 00:54:49.77	Datawindow retrieve start	SQLWrapper[SQLType=Select, SQL=SELECT "department"."dept_id", "department"."dept_name", "department"."dept_head_id" FROM "department" ORDER BY "department"."dept_id" ASC, SubSections=[SELECT "department"."dept_id", "department"."dept_name", "department"."dept_head_id" FROM "department" ORDER BY "department"."dept_id" ASC]]
2061427769	sqlca	2012-04-28 00:54:49.77	Datawindow retrieve end	✔ The Datawindow retrieve has been executed successfully.
2061427769	sqlca	2012-04-28 00:54:51.807	Embedded SQL start	SQLWrapper[SQLType=Insert, SQL=INSERT Into department (dept_id, dept_name, dept_head_id) Values (1000, 'ApeonTest', 100), SubSections=[INSERT Into department (dept_id, dept_name, dept_head_id) Values (1000, 'ApeonTest', 100)]]
2061427769	sqlca	2012-04-28 00:54:51.807	Embedded SQL end	✔ The Embedded SQL has been executed successfully.
2061427769	sqlca	2012-04-28 00:54:52.150	Embedded SQL start	SQLWrapper[SQLType=Select, SQL=SELECT dept_name From department Where dept_id = 100, SubSections=[SELECT dept_name From department Where dept_id = 100]]
2061427769	sqlca	2012-04-28 00:54:52.165	Embedded SQL end	✔ The Embedded SQL has been executed successfully.

Check to view the detailed information of the transaction, maybe you can figure out which SQL statements is blocked and then optimize you PB code for Apeon.

3.3 Identifying Performance Bottlenecks of Web Server and Application Server

You can use tools like Load Runner to identify the performance bottlenecks of your web server and application server. Usually you should consider checking these places: server runtime environment, network connectivity conditions, and performance relevant parameter settings (such as timeout settings and cache settings).

Or you can refer to the information provided by your web server and application server vendor, it will help you more accurately locate the performance bottlenecks and make appropriate settings according to your actual workload.

3.4 Identifying Performance Bottlenecks of DB Server

3.4.1 Deadlock analysis

If you are doing the stress test and you find some problems during the test, you should check if there are deadlock of the database before you quite the stress test mode. By the way, if you find the performance problem when doing the database related operation, you should also check the deadlock in database.

Take SQL Server database as an example, if you want to check whether there are deadlocks or not, you can use the following way:

- Execute sp_lock in SQL query analyzer. If you find there are 'X' or 'IX' of the Mode field, then it means there are deadlocks. You can find out the deadlocks occurred in which tables by the information in Resource field.
- Execute sp_who spid in SQL query analyzer, then you can find out the host name of spid and commands being executed.

3.5 Identifying Performance Bottlenecks of PB application

If your PB application includes poorly written SQL and extensive use of non-datawindow based (direct SQL statements) SELECTs, UPDATEs, DELETEs and INSERTs, of course your application runs slow. Tune the performance of PB application is the last but the most important step. Usually the performance can be obviously improved after modifying the codes of the PB application.

3.5.1 Analyzing performance bottlenecks of PB application

Output time log for the code and find out time-consuming ones during execution. Please refer to the following Scripts and load it into PowerBuilder Application. A log with time interval regarding code will be generated while the object of _log(string arg) is called.

Please check the start time and end time of the code and find out time-consuming ones.

```
string is_LogFile="C:\debug\logservice.log"
string is_workdir="C:\debug\"
long il_LogFileHandle
IF NOT directoryexists(is_workdir) THEN
  createdirectory(is_workdir)
END IF
il_LogFileHandle = FileOpen(is_LogFile,StreamMode!,write!,shared!,append!)
FileWrite(il_LogFileHandle,"~r~n~r~n")
of_log("====Log service initialized====")

public function integer of_log (string as_message);
IF FileExists(is_LogFile) THEN
  IF FileLength(is_LogFile) > 2097152 THEN//2M
    FileClose(il_logfilehandle)
    FileCopy(is_LogFile,is_LogFile+".bak",true)
    FileDelete(is_LogFile)
    il_LogFileHandle = FileOpen(is_LogFile,StreamMode!,write!,shared!,append!)
  END IF
END IF
as_message = "~r~n"+string(now(),"yyyy-mm-dd hh:mm:ss.fff") +as_message
FileWrite(il_logfilehandle,as_message)
return 1
end function
```

4 Tuning: DB Server

4.1 Database

The database performance will be improved from the following aspects:

- Choose proper database driver. Refer to [Recommended database driver](#) in *Database performance settings* for details.
- Set reasonable timeout settings. Refer to [Command Timeout](#) in *Recommended database setting* for details.
- Optimize the table structure.
- Use proper index.
- Optimize SQL statements.

5 Tuning: Excessive Server Calls

5.1 Overview

Excessive server calls in a given operation can create performance issues for that operation on slow and high-latency networks. If you are not familiar with the concept of "server calls", please refer to [Impact of the Internet and slow networks](#) and then proceed with this section.

This section will provide four different techniques including code examples to minimize server calls and thereby optimize the performance of your PowerBuilder application for a WAN or the Internet.

1. Partition transactions utilizing stored procedures
2. Partition non-visual logic utilizing server-side non-visual objects (NVOs)
3. Eliminating Recursive Embedded SQL
4. Group multiple server calls into one "group" call with Apeon Labels

5.2 Technique #1: partitioning transactions via stored procedures

Imagine your PowerBuilder client contains the following code:

```
long ll_rows, i
decimal ldec_price, ldec_qty, ldec_amount

ll_rows = dw_1.retrieve(arg_orderid)
for i = 1 to ll_rows
    dw_1.SetItem(i, "price", dw_1.GetItemDecimal(i, "price")*1.2)
next

if dw_1.update() < 0 then
    rollback;
    return
end if

for i = 1 to ll_rows
    ldec_price = dw_1.GetItemDecimal(i, "price")
    ldec_qty = dw_1.GetItemDecimal(i, "qty")

    if ldec_price >= 100 then
        ldec_amount = ldec_amount + ldec_price*ldec_qty
    end if
end if
Next

ll_rows = dw_2.Retrieve(arg_orderid)
dw_2.SetItem(dw_2.GetRow(), "amount", ldec_amount)

If dw_2.update() = 1 then
    Commit;
else
    rollback;
end if
```

This is not only problematic from a runtime performance perspective since there would be numerous server calls over the WAN, but also it could result in a "long transaction" that would tie up the database resulting in poor database scalability.

The business logic and the data access logic (for saving data) are intermingled. When the first "Update()" is submitted to the database, the related table in the database will be locked until the entire transaction is ended by the "Commit()". The longer a transaction is the longer other clients must wait, resulting in fewer transactions per unit of time.

To improve the performance and scalability of the application, the above code can be partitioned in two steps:

1. First, move the business logic (or as much possible) outside of the transaction. In other words, the business logic should appear either before all Updates of the transaction or after Commit of the transaction. This way the transaction is not tied up while the business logic is executing.
2. Second, partition the transaction whereby all the Updates are moved into a stored procedure. The stored procedure will be executed on the database side and only return the final result. This would eliminate the multiple server calls from the multiple updates to just one server call over the WAN for saving all the data in one shot.

It is generally best to actually divide the original transaction into three segments or procedures: "Retrieve Data", "Calculate" (time-consuming logic), and "Save Data". The "Retrieve Data" procedure retrieves all required data for the calculation. This data usually would be cached in a DataWindow(s) or a DataStore(s). In the "Calculate" procedure, the data cached in DataStore will be used to perform the calculation instead of retrieving data directly from the database. The calculation result would be cached back to a DataStore and then saved to the database by the "Save Data" procedure.

Example of the new PB client code partitioned into three segments and invoking a stored procedure to perform the Updates:

```
long ll_rows, i
decimal ldec_price, ldec_qty, ldec_amount
//Retrieve data
dw_2.Retrieve(arg_orderid)
ll_rows = dw_1.retrieve(arg_orderid)
//Calculate (time-consuming logic)
for i = 1 to ll_rows
    dw_1.SetItem(i, "price", dw_1.GetItemDecimal(i, "price")*1.2)
next

for i = 1 to ll_rows
    ldec_price = dw_1.GetItemDecimal(i, "price")
    ldec_qty = dw_1.GetItemDecimal(i, "qty")

    if ldec_price >= 100 then
        ldec_amount = ldec_amount + ldec_price*ldec_qty
    end if
Next

dw_2.SetItem(dw_2.GetRow(), "amount", ldec_amount)
//Save data
declare UpdateOrder procedure for up_UpdateOrder @OrderID = :arg_orderid,
@amount = :ldec_amount;
execute UpdateOrder;
```

Example of code for the stored procedure to Update the database:

```
create procedure up_UpdateOrder(
```

```
@orderid integer,  
@amount decimal(18, 2)  
)  
as  
begin  
update order_detail set price = price*1.2  
where ordered = @orderid  
  
if @@error <> 0  
begin  
    rollback  
    return dba.uf_raiseerror()  
end  
  
update orders set amount = @amount  
where ordered = @orderid  
  
if @@error <> 0  
begin  
    rollback  
    return dba.uf_raiseerror()  
end  
  
commit  
end
```

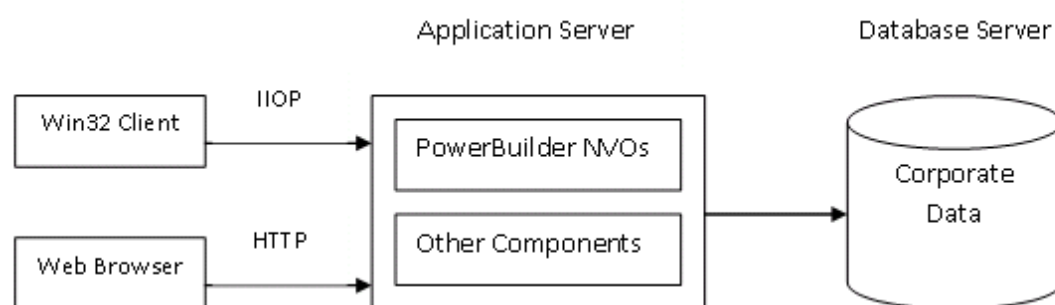
In summary, with the above performance optimization technique, the performance and scalability is improved since the transaction is shorter. The server call-inducing Updates are all implemented on the server-side rather than the client-side, improving the response time. Secondly, moving the business logic out of the transaction further shortens the transaction. If the business logic cannot be moved out of the transaction, one may want to consider implementing the business logic together with the transaction as a stored procedure. In summary, shorter transactions equals better scalability and faster performance.

5.3 Technique #2: partitioning non-visual logic via NVOs

Partitioning non-visual logic and encapsulating it within PowerBuilder NVOs has been a long-time best practice among PowerBuilder developers. What's relatively new, however, is utilizing middleware or application server, such as Microsoft IIS, IBM WebSphere*, Oracle WebLogic*, TmaxSoft JEUS*, or JBoss* to deploy these NVOs to the server (i.e. server-side NVOs) and invoke them from the client over IIOP or HTTP.

** Requires an SAP Sybase plug-in that is purchased separately from SAP or authorized resellers.*

To give you a better idea of what this looks like, the following diagram shows a very high-level architecture of PowerBuilder applications utilizing server-side NVOs when deployed as a Windows Client/Server application as well as an n-tier Web (.NET* or J2EE) application with Apeon PowerServer.

Figure 5.1: Architecture with server-side NVOs

Deploying your non-visual logic as server-side NVOs is an excellent way to boost the performance of your application over a WAN; however, some thought must be given as what to partition. On one hand, consolidating your business logic within NVOs will significantly thin out client-side processing and move all those server call-inducing statements to the server-side running in a low-latency LAN environment. On the other hand, each invocation of the server-side NVO is a server call in itself.

If the non-visual logic you are partitioning contains multiple statements that result in server calls, then this would be a good candidate. However, if your non-visual logic does not contain any statements that would result in server calls, then by partitioning this you have not only created more work for yourself but actually added an additional server call that didn't exist before. So unfortunately it's not as simple as moving all non-visual logic to the server-side.

Imagine your PowerBuilder client contains the following code:

```

long ll_id

dw_1.Retrieve()
dw_1.SetSort("#1 A, #2 D")
dw_1.Sort()

declare order_detail cursor for
select id from order_detail where orderid = :arg_orderid;
open order_detail;
fetch order_detail into :ll_id;

do while sqlca.sqlcode = 0
  update order_detail set price = price*1.2
  where orderid = :arg_orderid and id = :ll_id;

  if sqlca.sqlcode < 0 then
    rollback;
    return
  end if

  fetch order_detail into :ll_id;
loop
close order_detail;
commit;

dw_2.Retrieve()
dw_2.SetFilter("price >= 100")
dw_2.Filter()
  
```

The code in bold above would be good candidate for partitioning as server-side NVOs while the rest of the code should remain at the PB client. After partitioning this logic, the new PB client code, which would invoke the server-side NVO, would be as follows:

```

n_order ln_order
long ll_rc

dw_1.Retrieve()
dw_1.SetSort("#1 A, #2 D")
dw_1.Sort()

ll_rc = myconnect.CreateInstance(ln_order, "PB_pkg_1/n_order")
if ll_rc = 0 then
    ln_order.of_UpdateOrderPrice(arg_orderid)
end if

dw_2.Retrieve()
dw_2.SetFilter("price >= 100")
dw_2.Filter()

```

With this technique we have reduced those numerous server calls of the database transaction to just one single call to the NVO, and at the same time created a re-usable component that can be shared by other modules in our PowerBuilder application or shared by other applications.

5.4 Technique #3: eliminating recursive Embedded SQL

It's actually quite common to find Embedded SQL in a loop, especially Select and Insert statements. As explained previously, server calls that are recursive in nature are quite dangerous, potentially generating tremendous number of server calls. If your application requires loops or recursive functions, it would be best to replace any code resulting in server calls with code that does not.

For this technique, we will assume we have Select and Insert SQL statements in a loop. The general idea is to first create a DataWindow/DataStore using the SQL. Then replace the SQL statements contained in the loop with PowerScript modifying the DataWindow/DataStore, which does not result in server calls. If the SQL statement contained in the loop is an Insert statement, we would want to replace that with PowerScript that would insert data into the DataWindow/DataStore. Once all the data has been inserted, then in one shot we would update the DataWindow/DataStore to the database (outside the loop), resulting in only one server call. If the SQL statement contained in the loop is a Select statement, we would retrieve data into a DataWindow/DataStore before executing the loop, and then write PowerScript in the loop to select the desired data from the DataWindow/DataStore.

The following is a code example that increases the price of a specific order by 20%, where Embedded SQL is used to update the change row-by-row (hence the loop), and then save those changes to the database:

```

long ll_id

declare order_detail cursor for
select id from order_detail where orderid = :arg_orderid;
open order_detail;
fetch order_detail into :ll_id;

do while sqlca.sqlcode = 0
    update order_detail set price = price*1.2
    where orderid = :arg_orderid and id = :ll_id;

    if sqlca.sqlcode < 0 then
        rollback;
        return
    end if
end do

```

```
end if

  fetch order_detail into :ll_id;
loop
close order_detail;
commit;
```

Now we will replace the Embedded SQL with a DataWindow. Specifically, we will cache the data in a DataWindow and update the database with a single DataWindow Update, resulting in just once server call:

```
long ll_rows, i

ll_rows = dw_1.retrieve(arg_orderid)
for i = 1 to ll_rows
  dw_1.SetItem(i, "price", dw_1.GetItemDecimal(i, "price")*1.2)
next

if dw_1.update() = 1 then
commit;
else
rollback;
end if
```

With this technique we have just eliminated server calls from inside the loop, reduced the number of server calls to just one, and created a data caching mechanism at the client-side that can be used to feed data to other controls of the PowerBuilder client.

5.5 Technique #4: grouping multiple server calls with Apeon Labels

Part of this section, mainly including introduction to Apeon Labels and the two examples, are quoted from an article titled [Apeon Performance Tuning](#) published in [ISUG Journal - October 2011 Edition](#) written by [Yakov Werde](#).

PowerServer deployment is a multiphase process. In the first phase PowerScript code is analyzed and converted into two categories of code (1) HTML and JavaScript that interact with the Browser and (2) JavaScript that interacts with the Apeon ActiveX component. During this phase the deployment tool converts embedded SQL and DataWindow database centric code (retrieves and updates) into RPCs (remote procedure calls) that interact with PowerServer components in the Application Server. The important fact to understand is that both the PowerServer deployment and runtime engines distinguish between the different categories of code. Additionally, Apeon engineers, recognizing the importance of performance tuning RPC code, provided language constructs that allow an application developer to demarcate and group RPC code in ways that will allow the runtime engine to performance enhance browser/server communication. These performance enhancing language extensions are called Apeon Labels.¹

Apeon PowerServer provides seven "Apeon Label" functions, which can be found in the `apeon_nvo_db_update` object in `apeon_workaround.pbl` (Although the PBL is named `workarounds`, it also contains many useful utility classes and methods). When you examine the code of these functions, you will find the code either looks like the same nested

¹Quoted from an article titled [Apeon Performance Tuning](#) published in [ISUG Journal - October 2011 Edition](#) written by [Yakov Werde](#).

PowerScript you have already written (such as `of_update()`), or has no implementation (such as `of_startqueue()`). The fact is: Apeon Label functions serve as markers (hence, the name Label) to the deployment tool; and during the deployment process, Apeon generates efficient JavaScript ActiveX component call code that implements bandwidth saving functionality in the Web tier client. Therefore, Apeon Labels do not execute any code or modify how your PowerBuilder application works in a Client/Server environment. Rather, when used on the Web it will notify Apeon's runtime Web libraries to handle certain database operations differently than PowerBuilder with the aim of reducing the number of server calls.

Below are descriptions of the seven "Label" functions about how they handle the database operation. For details about the syntax and return values of these functions, please refer to Section 1.3.5, "Apeon Labels" in *Workarounds & APIs Guide*.

Table 5.1: Apeon Label functions

Label	Function	Description
Commit/ Rollback Label	<code>of_autocommitrollback</code>	Notifies the PowerServer Web application to automatically commit or roll back the first database operation statement after the label.
Commit Label	<code>of_autocommit</code>	Notifies the PowerServer Web application to automatically commit the first database operation.
Rollback Label	<code>of_rollback</code>	Notifies the PowerServer Web application to automatically roll back the first database operation statement if the operation fails.
Queue Labels (Consists of Start Queue Label and Commit Queue Label)	<code>of_startqueue</code> <code>of_commitqueue</code>	These two labels must be used in pairs. They notify the PowerServer Web application not to commit database operations after the Start Queue Label until the Commit Queue Label is called (and unless an Apeon Immediate Call Label is called).
Immediate Call	<code>of_imdcall</code>	Notifies the PowerServer Web application to immediately commit a database operation.
Update Label	<code>of_update</code>	It is used to reduce the number of interactions with the server caused by "interrelated updates". "Interrelated updates" usually occurs when the update result of one DataWindow determines whether another DataWindow should be updated.

Apeon Commit/Rollback Label

The Apeon Commit/Rollback Label (`of_autocommitrollback`) notifies Apeon to automatically commit or roll back operations to the database after updating or inserting.

For example:

```
gmv_apeonDbLabel.of_AutoCommitRollback()
update tab_a .....
if sqlca.sqlcode=0 then
```

```

..... //code independent of database operations
commit;
.....
else
.....
rollback;
.....
endif

```

Appeon Commit Label

The Appeon Commit Label (`of_autocommit`) notifies Appeon to commit operations to database after updating and inserting.

For example:

```

gmv_appeonDbLabel.of_AutoCommit()
update tab_a .....

```

Appeon Queue Labels

There are two Appeon Queue Labels, the Start Queue Label (`of_startqueue`) and the Commit Queue Label (`of_commitqueue`). These two labels must be used in pairs. They notify Appeon not to commit database operations after the Start Queue Label until the Commit Queue Label is called (and unless an Appeon Immediate Call Label is called).

For example:

```

gmv_appeonDbLabel.of_StartQueue()
dw_1.retrieve(arg1,arg2)
dw_2.retrieve(arg3,arg2)
.....
dw_3.retrieve(arg4)
gmv_appeonDbLabel.of_CommitQueue()

```

Appeon Immediate Call Label

The Appeon Immediate Call Label (`of_imdcall`) is used between the Appeon Start Queue Label and Appeon Commit Queue Label, when the return value of an operation that is called after the Appeon Start Queue Label determines the subsequent business logic, for example, the return value is used in a CASE or IF...THEN expression.

For example:

```

gmv_appeonDbLabel.of_StartQueue()
dw_1.retrieve()
gmv_appeonDbLabel.of_ImdCall()
select ... into :var_1,:var_2 .....

if var_1>0 then
  para = "ok"
else
  para = "false"
end if
dw_2.retrieve(para)
gmv_appeonDbLabel.of_CommitQueue()

```

Appeon Update Label

The Appeon Update Label (`of_update`) is used to reduce the number of interactions with the server caused by "interrelated updates". "Interrelated updates" usually occurs when the update result of one DataWindow determines whether another DataWindow should be updated.

The following example shows how Apeon uses the Update Label to reduce client-server interactions:

Example of interrelated updates:

```
if dw_1.Update()=1 then
  if dw_2.Update()=1 then
    commit;
    MessageBox("Success","Update success!")
  else
    rollback;
    MessageBox("Failure","Update all failure!")
  end if
else
  rollback;
  MessageBox("Failure","Update dw_1 failure!")
End if
```

Use the Apeon Update Label to rewrite the example:

```
l_rtn = gnv_apeonDb.of_Update(dw_1,dw_2)
if l_rtn=1 then
  MessageBox("Success","Update success!")
elseif l_rtn= -102 then
  MessageBox("Failure","Update all failure!")
Else
  MessageBox("Failure","Update dw_1 failure!")
End if
```

Script defined in the Update Label associated function, of_Update(dw_1,dw_2):

```
if dw_1.Update()=1 then
  if dw_2.Update()=1 then
    commit;
    return 1
  else
    rollback;
    return -102
  end if
else
  rollback;
  return -101
end if
```

The more database operations utilize Apeon Labels, the faster the performance will be. For PowerBuilder applications deployed to the Web with Apeon PowerServer, in many cases you will achieve acceptable runtime performance simply by utilizing this technique. The reason is that there are a number of features built into Apeon's infrastructure framework that automatically boost the performance of PowerBuilder applications over the Web. The performance boosting features are discussed in [Automatic performance boosting](#).

Two Examples¹

Now, let's see two situations where Apeon Labels reduce network traffic.

The first example illustrates performance enhancements when chaining DataWindow updates. Oftentimes, data contained in multiple data controls (datawindow, datastore or childdatawindow) must be grouped into a logical unit of work (LUW).

The following figure shows the pseudocode update algorithm. This code necessitates three browser to server round trips, one for each data control.

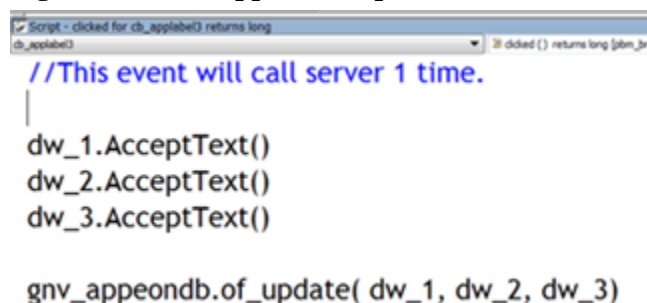
Figure 5.2: Nested Update Algorithm

```

|
IF dw_1.update( true, false) then
  IF ds_1.update( true, false) then
    IF dwc_1.update( ) then
      commit;
    ELSE
      rollback; //dwc failed
  ELSE
    rollback; //ds failed
ELSE
  rollback; //dw failed
END IF

```

(Apeon groups all changed data into a single transmission). As you can see from the following figure, refactoring the code to use the Apeon Label of `_update()` method, reduces three round trips one. In the event of a failed update the method returns a result code indicating which update failed.

Figure 5.3: Call Apeon of `_update` method()


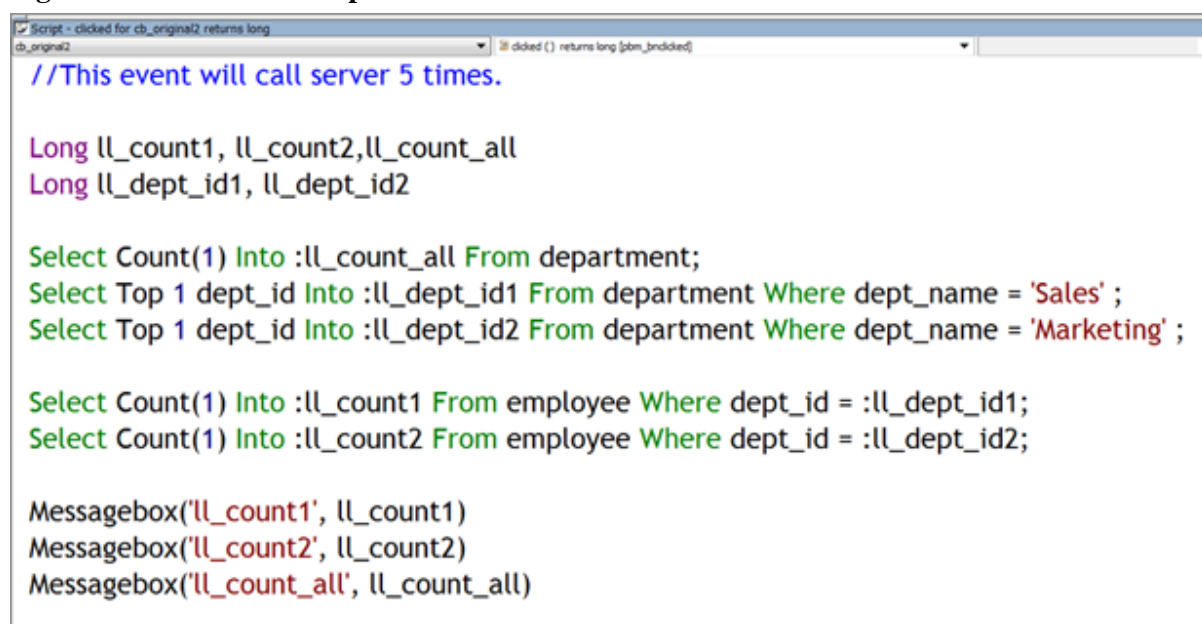
```

Script - clicked for cb_applabel() returns long
cb_applabel() | clicked () returns long [pbm_br]
//This event will call server 1 time.
|
dw_1.AcceptText()
dw_2.AcceptText()
dw_3.AcceptText()

gnv_apeondb.of_update( dw_1, dw_2, dw_3)

```

Sometimes a script has multiple embedded SQL statements grouped into a single LUW. Each SQL statement causes a browser to server round trip. The following figure illustrates one such (simplified) scenario.

Figure 5.4: Five Round Trips


```

Script - clicked for cb_original2 returns long
cb_original2 | clicked () returns long [pbm_binded]
//This event will call server 5 times.

Long ll_count1, ll_count2, ll_count_all
Long ll_dept_id1, ll_dept_id2

Select Count(1) Into :ll_count_all From department;
Select Top 1 dept_id Into :ll_dept_id1 From department Where dept_name = 'Sales' ;
Select Top 1 dept_id Into :ll_dept_id2 From department Where dept_name = 'Marketing' ;

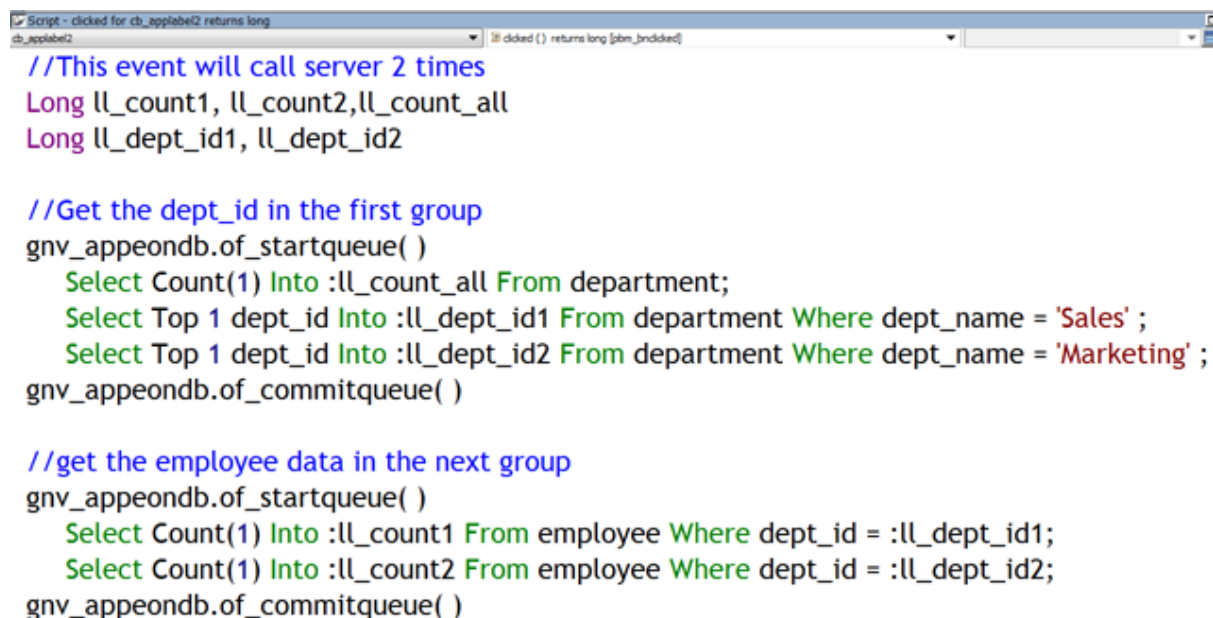
Select Count(1) Into :ll_count1 From employee Where dept_id = :ll_dept_id1;
Select Count(1) Into :ll_count2 From employee Where dept_id = :ll_dept_id2;

MessageBox('ll_count1', ll_count1)
MessageBox('ll_count2', ll_count2)
MessageBox('ll_count_all', ll_count_all)

```

In this case you can use the Apeon Labels of `of_startqueue()` and `of_endqueue()` to demarcate logical statement groupings thereby reducing the number of server round trips. The code shown in the following figure reduces the number of server round trips by over 50% by dividing the SQL into two logical groups; the first group of statements acquires values necessary for statements in the second group.

Figure 5.5: Reduced to 2 Round Trips



```
Script - clicked for cb_applabel2 returns long
cb_applabel2
clicked () returns long [pbm_bndclicked]

//This event will call server 2 times
Long ll_count1, ll_count2, ll_count_all
Long ll_dept_id1, ll_dept_id2

//Get the dept_id in the first group
gnv_appeondb.of_startqueue( )
  Select Count(1) Into :ll_count_all From department;
  Select Top 1 dept_id Into :ll_dept_id1 From department Where dept_name = 'Sales' ;
  Select Top 1 dept_id Into :ll_dept_id2 From department Where dept_name = 'Marketing' ;
gnv_appeondb.of_commitqueue( )

//get the employee data in the next group
gnv_appeondb.of_startqueue( )
  Select Count(1) Into :ll_count1 From employee Where dept_id = :ll_dept_id1;
  Select Count(1) Into :ll_count2 From employee Where dept_id = :ll_dept_id2;
gnv_appeondb.of_commitqueue( )
```

6 Tuning: Heavy Client

6.1 Overview

If you find your PowerServer Web application performs poorly even when run in a local environment, then chances are the JavaScript interpreter of the Web browser is slowing you down. Generally, a newer or more high-power computer will clear up many situations. But if it is not an option to utilize a late model computer or if the performance issue still persists, then you would want to consider optimizing your PowerBuilder code to make it more efficient when running in the Web browser. This section provides several different techniques for dealing with several specific types of inefficient PowerBuilder coding practices.

6.2 Technique #1: thin-out #heavy# Windows

Redesign the navigation strategy to present a lighter-weight Client user interface. Specifically, you would want to focus on reduce the number of DataWindows and DropDownDataWindows in a particular window or tab page. In many situations, the DataWindows in a Window can be spread out across multiple Windows or tabs, thereby reducing the "weight" of the Window. It may be possible to rework your DropDownDataWindows as DropDownListBoxes. By thinning out the UI, it will not only make the Window run faster but your users will not be overwhelmed by so much data.

6.3 Technique #2: thin-out #heavy# UI logic

This section is broken down into several subsections, utilizing the same technique to deal with various types of "heavy" UI logic.

6.3.1 Manipulating the UI in loops

Excessive and unnecessary loops have a negative impact on performance. Some PowerBuilder code will trigger your PowerServer Web application to redraw visual objects, such as DataWindows, controls, etc. If such functions are put into a loop, it will redraw the visual objects numerous times and therefore negatively affecting performance.

PowerServer recommends the following:

- Do not put functions that operate on DataWindow rows into a loop.
- Avoid placing functions that result in the repaint of visual control(s) into a loop; otherwise, the visual control(s) will be repainted many times while the loop is executed.
- Use the Find function for DataWindow search instead of using the loop statement.

The following is an example:

```
Long ll_row
String ls_expression
String ls_Name
ls_Name = "Mike"
For ll_row = 1 To dw_1.RowCount()
    ls_expression =
        dw_1.GetItemString(ll_row,"name")
    If ls_expression = ls_Name Then
```

```
Exit
End If
Next
```



```
Long ll_row
Long ll_rowcount
String ls_expression
String ls_Name
ls_Name = "Mike"
ll_rowcount = dw_1.RowCount()
For ll_row = 1 To ll_rowcount
  ls_expression = +
  dw_1.GetItemString(ll_row, "name")
  If ls_expression = ls_Name Then
    ...
  Exit
End If
Next
```



```
Long ll_row
Long ll_rowcount
String ls_expression
String ls_Name
ls_Name = "Mike"
ll_rowcount = dw_1.RowCount()
ls_expression = "name = '" + ls_Name + "'"
ll_row = dw_1.Find(ls_expression, 1, ll_rowcount)
...
```

6.3.2 Triggering events repeatedly

Frequent triggering of events such as Timer, MouseMove, and SelectionChanging slows down performance. For example, once a Timer event is triggered, it occurs repeatedly at a specified interval that can be set to even milliseconds (1/1000 of a second). Minimize the usage of events with high repetition such as Timer, MouseMove, SelectionChanging, GetFocus, LoseFocus, Activate, and Deactivate.

6.3.3 Performing single repetitive tasks

Use batch operations instead of performing a single operation many times. For example, the execution of the following "batch" code is two to three times faster than the original code.

```
For I = 1 To 100
  dw_1.SetItem(ll_i, +
               "name", +
               dw_2.GetItemString(ll_i, "name"))
  ...
Next
```



```
dw_1.RowsCopy(1, 100, Primary!, dw_2, 1, 100, Primary!)
```

6.3.4 Initializing #heavy# tabs

For windows containing Tab controls, if the Tab control contains more than five tab pages or the initialization of each tab page is complex, PowerServer recommends you use the CreateOnDemand method of Tab control to improve the runtime performance of these windows.

When CreateOnDemand is enabled, only the current tab page that is created will be initiated. The initialization of the hidden tab pages takes place only when they are selected. Therefore the window will operate more efficiently.

Please keep in mind that as you make this change you may also need to modify other code of the Tab control. For example, if the current tab page uses data from another page, you need to:

1. Move the script that is used for obtaining data, to the SelectionChanged event.
2. Add a condition to validate whether the tab page carrying the data has been initiated. If initiated, the current tab page will successfully obtain the data; if not initiated, the user must select the tab page for initialization purposes, and the current tab page will successfully obtain the data.

6.3.5 Using ShareData or RowsCopy/RowsMove for data synchronization

The following PowerBuilder functions can synchronize data between DataWindows: ShareData, RowsCopy/RowsMove, Object.Data, and SetItem. ShareData is the fastest and it is recommended to use it whenever you need to synchronize data between DataWindows. SetItem is the slowest and should be avoided as much as possible. If ShareData cannot be or should not be used for some reason, then consider using RowsCopy/RowsMove followed by Object.Data.

6.3.6 Using computed fields

Computed fields involve a lot of recalculation in many situations; for example, when a column is deleted, added, or renamed. This recalculation is a process-intensive task, which negatively impacts performance and can be worked around. Therefore, PowerServer recommends the following:

- Avoid using computed fields in detail bands. Instead, add expressions in the SQL statements for getting specific data.
- Avoid embedding a computed field in an existing computed field.
- If a computed field is "Text: Sum or Expression", it is recommended that you divide the column into two columns: an edit style column with the "Text", and a computed field with "Sum or Expression".

6.3.7 Using DataWindow expressions

Generally speaking, DataWindow expressions will slow-down the initial display or subsequent refresh of DataWindows. As such, we recommend you reduce the usage of DataWindow expressions if possible, especially in the following situations:

- Avoid using DataWindow expressions for computing and setting column properties.
- Avoid setting sort and filter criteria directly for a DataWindow object. Instead, write the sort and filter criteria in the SQL statement of the DataWindow object. As noted previously, it is faster to use SQL statements than DataWindow functionality.

6.3.8 Using complex filters

Filters are considered "complex" if the filter criteria contain one or more expressions that call to one or more functions. It is recommended that you not use complex filtering on a DataWindow, especially on a DataWindow that has large amounts of data.

6.3.9 Using RowsFocusChanging/RowsFocusChanged events

The DataWindow RowsFocusChanging and RowsFocusChanged events can be triggered under many situations, especially when a DataWindow retrieves data. Since data is usually automatically retrieved into DataWindows when a Window is opened, if a lot of code is written into the RowsFocusChanging and RowsFocusChanged events, it will significantly prolong the time it takes to open the Window and display the DataWindow. PowerServer recommends that you do not write code into RowsFocusChanging and RowsFocusChanged events unless it is necessary.

6.4 Technique #3: offload #heavy# non-visual logic

Instead of trying to write "heavy" logic more efficiently or avoiding use of "heavy" logic, the simplest way is just to offload all that "heavy" logic to the application server, which is designed to handle the most daunting tasks. The only catch is that only non-visual logic can be run at the application server.

The following types of non-visual logic can be encapsulated in PowerBuilder NVOs and deployed to the application server to eliminate "heavy" logic from the Web browser:

- Complex non-visual events and functions, especially non-visual events and functions that contain dynamic SQL, Cursor statements, Stored Procedure calls, and other SQL statements.
- Validation of updated data.
- A series of data computations or a complex data computation.

7 Tuning: Large Data Transmissions

This section introduces two common-used techniques to reduce the size of data transmission:

7.1 Overview

Suppose you have worked hard to make an application Web-ready using Apeon, and, using your test data, it seemed to perform acceptably. Then, when your users provide "live" test data in realistic volumes, you discover that the application takes a long time to load, and worse, a long time to respond to your user's input. What to do?

Well first you should confirm that your issue is not being caused by excessive server calls (see [Tuning: Excessive Server Calls](#)). The reason is that majority of the time, PowerBuilder applications are coded such that as additional rows of data are retrieved logic is executed to validate, manipulate, or otherwise handle the data, which can result in server calls. As such, the more rows of data are retrieved the more server calls are made.

Once you are certain the slow-down is not caused by excessive server calls then you can consider reducing the size of data transmission. So what can do practically? Well at a high-level there are several techniques you can employ:

- The first and most popular is staging the data retrieval into manageable increments. For example, you can expose a Next button, and have the application respond to this button click by getting the next logical segment of the result set just like typical Websites or Web applications. [Technique #1: retrieving data incrementally](#) gives you instructions on how to achieve this.
- Another technique is to create multiple smaller "specific" views rather than one larger "general" view. Consider adding SQL WHERE clauses based on more search criteria, thus retrieving only the amount of data that is absolutely necessary for a particular view of interest.
- If you have a choice between reducing the number of rows retrieved, and reducing the number of columns, note that a small reduction in columns (described below in [Technique #2: minimizing excessive number of columns](#)) can improve performance to an even greater extent than a reduction in rows. This is because most of the time, loops, whether in the application code or in the virtual machine, visit columns first and then rows.

Anything you do to reduce the size of the result set in one way or another can only improve performance and possibly improve usability of your application as well.

7.2 Technique #1: retrieving data incrementally

7.2.1 For Oracle database server

Oracle includes a pseudo-column called ROWNUM which allows you to generate a list of sequential numbers based on ordinal row. If your application uses Oracle database, apply your Oracle skills and ROWNUM to limit the number of returned rows. For example, this query selects the TOP 10 rows from a table:

```
SELECT *
```



```
FROM (SELECT * FROM my_table ORDER BY col_name_1)
WHERE ROWNUM BETWEEN 1 AND 10;
```

You can impose a NEXT button to the DataWindow. In the Clicked event of the NEXT button, the query changes with ROWNUM increments by 10. Therefore, when the NEXT button is clicked, the DataWindow displays next 10 rows.

7.2.2 For all other database servers

If your application uses a non-Oracle database (for example, Microsoft SQL server) you can use the following SQL syntax to limit the number of returned rows to the DataWindow:

```
SELECT TOP 10 *
FROM my_table
WHERE Table.primary_key > = bottom
ORDER BY Table.primary_key;
```

"bottom" is a variable that contains the row number of the first row you want to retrieve, where rows are ordered by the primary key for the table. Before retrieving the first page of data, "bottom" should be set to a value smaller than any primary key value in the table.

Based on this SQL statement, you can implement Next and Previous buttons for the DataWindow. Their Clicked events increment or decrement the bottom variable so that its value matches the primary key value in the first row you want to retrieve then execute the above SQL statement.

7.3 Technique #2: minimizing excessive number of columns

As the number of rows in the result set increased, the number of columns will cause greater degradation on performance, especially for nested loops in your application which process rows in the outer loop, and columns in the inner loop. Sometimes the excessive number of columns is intentional and other times it is unintentional.

A sign of unintentionally excessive columns would be the SQL syntax Select * From: consider modifying this syntax to Select fieldList From, where fieldList is the comma-separated list of all, and only, those fields your application will actually need. The performance of the SQL syntax using asterisk will be automatically degraded any time your database administrator modifies the database design by adding columns.

A sign of intentionally excessive columns is simply a long list of columns in your SQL Select statement. Consider analyzing your actual needs to make certain all columns are necessary. It may be possible to request certain columns (needed only in exceptional circumstances) in a separate SQL operation. Please keep in mind if the Visible property of a column is set to zero (the control is not visible), even though the Column cannot be seen, it is still impacting performance.

8 Conclusion

PowerBuilder applications that perform well today in your local network may not perform well in a distributed architecture tomorrow. Likewise, typical PowerBuilder development practices may not be suitable for a distributed architecture. The several techniques outlined in this guide are intended to steer you in general directions. It is recommended to extrapolate from these examples and apply to your particular situation. Please keep in mind that excessive server calls is the single biggest culprit of performance issues over the Internet, which is a relatively high latency connection.

Purchasing expensive network connectivity and faster hardware can make up for suboptimal code. Sometimes the cost of doing this is less than the cost of optimizing the code. If you do take this route, keep in mind that a low-latency network connection is generally the key rather than a high-bandwidth connection. Reason being, for most PowerBuilder applications and deployments, it is the network latency that kills the runtime performance not bandwidth limitations.

Index

A

- advanced thread settings
 - execution Timeout, [13](#)
 - maxconnection, [13](#)
 - maxWorkerThreads and maxIoThreads, [12](#)
 - minFreeThreads and minLocalRequestFreeThreads, [13](#)
 - minWorkerThreads, [12](#)
 - recommended thread settings, [13](#)
- AEM performance settings, [8](#)
 - Custom Libraries download settings, [10](#)
 - DataWindow data caching, [10](#)
 - Log file settings, [10](#)
 - Multi-thread download settings, [10](#)
 - Timeout settings, [8](#)
- Analyzing log files
 - Analyzing log files, [16](#)
- Apeon performance
 - automatic performance boosting, [6](#)
 - expected performance level, [6](#)
 - impact of heavy client-side logic, [4](#)
 - impact of large data transmission, [5](#)
 - impact of the Internet and slow networks, [3](#)
- automatic performance boosting, [6](#)

C

- Custom Libraries download settings, [10](#)

D

- DataWindow data caching, [10](#)

E

- eliminate recursive Embedded SQL, [25](#)
- excessive server calls
 - eliminate recursive Embedded SQL, [25](#)
 - group multiple server calls with Apeon Labels, [26](#)
 - partition non-visual logic via NVOs, [23](#)
 - partition transactions via stored procedures, [21](#)
- execution Timeout, [13](#)
- expected performance level, [6](#)

G

- group multiple server calls with Apeon Labels, [26](#)

H

- Heavy Client
 - offload heavy non-visual logic, [35](#)
 - thin-out heavy UI logic, [32](#)
 - thin-out heavy windows, [32](#)

I

- identify performance bottlenecks
 - Identifying Performance Bottlenecks of PB application, [19](#)
- identifying performance bottlenecks
 - Analyzing performance bottlenecks of PB application, [19](#)
 - Deadlock analysis, [18](#)
 - identifying Performance Bottlenecks of DB Server, [18](#)
 - Identifying Performance Bottlenecks of Web Server and Application Server, [18](#)
- IIS server performance settings
 - advanced thread settings, [12](#)
 - recommendations for avoiding common errors, [11](#)
- impact of heavy client-side logic, [4](#)
- impact of large data transmission, [5](#)
- impact of the Internet and slow networks, [3](#)
- Impacts to Apeon performance, [3](#)
 - Impacts to Apeon performance, [3](#)
- initialize heavy tabs, [34](#)
- Internet Explorer performance settings, [10](#)

L

- large data transmissions
 - minimize excessive number of columns, [37](#)
 - retrieve data incrementally, [36](#)
- Log file settings, [10](#)

M

- manipulate the UI in loops, [32](#)
- maxconnection, [13](#)
- Maximum Rollback Retries, [9](#)
- maxIoThreads, [12](#)
- maxWorkerThreads, [12](#)
- minFreeThreads, [13](#)
- minimize excessive number of columns, [37](#)
- minLocalRequestFreeThreads, [13](#)
- minWorkerThreads, [12](#)
- Multi-thread download settings, [10](#)

O

offload heavy non-visual logic, [35](#)
Oracle database server, [36](#)
other database servers, [37](#)

P

partition non-visual logic via NVOs, [23](#)
partition transactions via stored procedures,
[21](#)
perform single repetitive tasks, [33](#)
performance settings
 AEM, [8](#)
 Internet Explorer, [10](#)
 PowerServer Toolkit, [8](#)
 Web and application server, [11](#)
PowerServer Toolkit performance settings, [8](#)

R

recommendations for avoiding common
errors on IIS, [11](#)
recommended thread settings, [13](#)
retrieve data incrementally
 Oracle database server, [36](#)
 other database servers, [37](#)
Rollback Completion time, [9](#)

S

Session timeout, [9](#)
Session Timeout Detection Interval, [8](#)

T

thin-out "heavy" UI logic
 perform single repetitive tasks, [33](#)
thin-out heavy UI logic, [32](#)
 initialize heavy tabs, [34](#)
 manipulate the UI in loops, [32](#)
 trigger events repeatedly, [33](#)
 use complex filters, [35](#)
 use computed fields, [34](#)
 use DataWindow expressions, [34](#)
 use RowsFocusChanging/
 RowsFocusChanged events, [35](#)
 use ShareData or RowsCopy/RowsMove
 for data synchronization, [34](#)
thin-out heavy windows, [32](#)
Timeout settings
 Maximum Rollback Retries, [9](#)
 Rollback Completion time, [9](#)
 Session timeout, [9](#)

 Session Timeout Detection Interval, [8](#)
 Transaction timeout, [9](#)
 Transaction Timeout Detection Interval, [9](#)
Transaction timeout, [9](#)
Transaction Timeout Detection Interval, [9](#)
trigger events repeatedly, [33](#)
Tuning: DB server
 Database, [20](#)

U

use complex filters, [35](#)
use computed fields, [34](#)
use DataWindow expressions, [34](#)
use RowsFocusChanging/
RowsFocusChanged events, [35](#)
use ShareData or RowsCopy/RowsMove for
data synchronization, [34](#)

W

Web and application server performance
settings
 Microsoft IIS server, [11](#)