Designing and Implementing Applications for Performance

Alfred J. Barchi

ajb@ajbinc.net
http://www.ajbinc.net/

Introduction

This paper describes a set of 'Best Practices' to for use in requirements analysis, design and implementation applications with the goal of high performance.

General Performance Requirements Analysis Questions

Performance requirements are driven by the needs of the customer. Performance is constrained by the capabilities of the hardware, software and infrastructure that host the application, as well as the architecture of the application. Customer satisfaction is a highly subjective thing that is difficult if not impossible to quantify (we could measure complaints per week, for example, but it would be difficult to establish an acceptable limit); however, customer satisfaction or lack thereof does drive changes to the application, so it also drives performance requirements.

Chances are that the new application is replacing an existing process, at least partially with the expectation that the new application will improve workflow, reduce overhead or increase profitability – all of which improve the bottom line. With this in mind, it makes sense to use an economic approach to analyzing performance requirements. Bigger, faster, more powerful applications tend to cost more to build and maintain, so there are cost tradeoffs to be considered.

For a functional baseline, we want to define requirements in terms of the end-user. If the user interfaces to the application through a web browser, for example, we want to define performance in terms of time to render the results of various operations to the user.

Customer Expectations

Customer satisfaction is driven by customer expectations, which in turn are driven by past experience.

- What process is the new application replacing? What are the performance parameters of the old process? What are the data requirements of the old process?

- What are the differences in functionality between the old process and the new application?

For a web-based application, we can define operations in terms of the amount of computational work required to perform the operation. Display of a static web page would be a computationally simple function.

Login and establishment of an SSL session, on the other hand, would be a moderately complex operation, involving exchange of protocol information, negotiating the algorithms to be used, and identification and authentication of the user. Large database operations are examples of computationally complex operations that can consume considerable time.

- For web applications, what are the key user operations (i.e., scenarios)? What kind of performance did the users experience in the past?

- Can we categorize user interactions into three or four classes based on their complexity?

- Can we categorize user interactions based on the frequency with which they are performed?

- Are there any constraints on the user interactions, e.g., session timeouts, operation timeouts, etc.?

Physical Constraints
Performance is constrained by physical limitations, e.g., bandwidth, processing power, storage capacity, etc., by physical constants, e.g., the speed of light, propagation delays, etc., by legal constraints, and by other applications contending for the same resources.

- Where is the application being hosted? Where are the users? What are the propagation delays?

- What is the available bandwidth, both in terms of processor bandwidth and in terms of communications bandwidth?

- What are the data capacity requirements? What are the performance impacts of scaling larger or smaller? Does the nature of the data make it amenable to a distributed approach versus a centralized approach?

- What are the applicable laws of the jurisdiction where the application is hosted, where the data is maintained, and through which the data passes? Do these laws constrain performance (e.g., laws and policies regarding encryption, logging, data retention, reporting, etc.)?

- What other applications contend for resources where the application is hosted, where the data is stored, and where the users interface, e.g., virus scanners, database maintenance operations, other applications, etc.?

External Constraints
Performance requirements are also driven by external constraints. There are only 24 hours in a day, and customers have other demands on their time. This may place constraints on the amount of time they have available to devote to using an application.

Additionally, there may be deadlines that an application must meet in order to be successful. For example, at a newspaper, copy must be received by the editor by T-x hours (where T is the time to go to press), delivered to layout by T-y hours, etc.

- What external constraints does the application have?

Mindset
It's virtually impossible to define performance requirements in terms of user satisfaction. The reason for this is that requirements, of necessity, must be verifiable. This means that they must be definable in quantifiable terms and be measurable.

Additionally, they must be translatable into a format that drives system design. For example, one could write a functional baseline requirement such as, "When queried as to the adequacy of the application's performance, a corporate officer shall reply to the effect that, 'Well, it's OK.'" This requirement is certainly verifiable – simply survey the corporate oficers who have used the application. Unfortunately, "it's OK" is a vague, subjective evaluation that nobody can design to. It requires a great deal of refinement and analysis to define the meaning of such things in concrete terms.

Unfortunately, this analysis also involves the concept of equity. It might not matter to the corporate officer if the new application runs ten times faster than the old one, if the new application achieves this performance improvement by omitting a capability that the he previously relied upon. It also might not impress the him if it costs him 20 times as much to achieve a tenfold increase in performance.

The key here is to recognize that while it is always important to design an application with performance in mind, there are applications where the concept of "good enough" simply doesn't apply.

We can, however, always evaluate an application's performance relative to the process that it replaces, and make tradeoffs in terms of new features, cost, etc., versus relative performance improvements. It simply requires recognizing when it is necessary to apply this approach versus when it isn't.

Use of Abstraction Layers
There is a trend these days toward using third-party solutions for implementing all or part of an application. We refer to these solutions as "abstraction layers" because they hide underlying complexity and present logical a view of the application from the perspective of the "problem domain."

Each layer of abstraction or indirection that is added to an application comes with a price. That price includes added complexity which affects maintainability, learning-curve issues, third-party dependencies and performance.

Sometimes, the price is worth paying. For instance, there are tools for performing symbolic mathematical operations, such as Maple and Mathematica, that provide an entire environment for such operations which enable users with mathematics skills but little or no programming skills to use them. These tools allow the user to "solve the problem in the problem domain."

An application can also utilize layers of indirection, such as a data access layer, to make the physical location of data opaque to the application in order to enhance scalability. For each layer of abstraction or indirection that is added to an application, the following questions must be asked:

- Does an abstraction layer significantly simplify the process of implementing the solution to the problem being addressed by the application?

- Is the solution to the problem being solved expected to evolve dynamically over time?

- Is the application intended to be a general purpose tool that can be used to solve an entire class of problems?

- Does the proposed abstraction layer provide any other significant benefits? For example, an abstraction layer may come in the form of a third party tool that provides pre-packaged interfaces with other systems, metrics collection, graphing functionality, image-processing functionality, etc.

Use of an abstraction layer necessitates that the developers who implement and maintain the application understand how to use the abstraction layer. This involves a learning curve, not only for the initial developers, but for anyone added to the project later on to maintain the application. This learning curve costs money.

Use of an abstraction layer adds to the application the built-in constraints and limitations of the abstraction layer. As the functionality of an application is evolved and upgraded over time, these constraints and limitations, which may not have been apparent when the abstraction layer was first chosen, can have a significant impact on the maintainability of the application, including the performance of new functionality.

Use of an abstraction layer provided by a third party carries with it a list of “third-party” issues, including licensing and distribution, third-party maintenance and upgrades (e.g., how will migrating from .NET 1.1 to .NET 2.0 impact us?), bug fixes, will the provider of the third party abstraction layer remain in business for the entire life-cycle of the application and will the abstraction layer be maintained for the entire life-cycle of the application.

Use of an abstraction layer involves a performance impact. Abstraction layers are, of necessity, intended to solve a class of problems by allowing the solution to be implemented in the problem domain. In other words, they are general purpose tools.

Typically, they involve the use of a special-purpose language that facilitates implementation of the solution. This language is then compiled, either into some form of byte-code that is executed by a run-time interpreter that runs either locally or on a server (such as Java), or into machine-generated source-code in a general-purpose language that is then compiled into the machine-code of the host machine (such as MSIL).

In either case, the special-purpose language will be designed for the general case and will require an underlying structure to be applied in order for the elements of the generated code to work together – a structure that is necessary for the tool to work properly but which has nothing to do with the end application itself.

This means that you will be hard-pressed to develop a tool using an abstraction layer that is as efficient as a special-purpose solution developed in a general-purpose programming language (at least theoretically). And, of course, the abstraction layer may come with scalability and stability issues. So there is a another cost tradeoff to consider.

It is important to be able to evaluate the performance of an abstraction layer. In order to do this, we might develop a test harness (i.e., a small test application or set of test applications) that performs functions similar to the ones that we expect our solution to perform, and in a similar ratio, i.e., ‘x’ number of searches, ‘y’ number of selects, ‘z’ number of updates, ‘a’ number numerical integrations, etc. Then we can load-test it to measure its performance at various levels of load, and stress-test it to identify any stability issues.

Horizontal scaling issues (i.e., problems, such as concurrency issues, caused by running the application simultaneously on multiple servers), could be identified through analysis using the test harness as an example.

This still begs the question of how to evaluate the performance impact of using the abstraction layer versus a direct implementation. Again, we can develop a test harness based on a direct implementation. If we find that it is too difficult or costly to do this, then that is a good indication that the abstraction layer is worth using. The question then becomes one of which implementation of the abstraction layer to use, e.g., ‘Product A’ or ‘Product B’.

Use of Indirection Layers

Indirection layers are similar to abstraction layers in that they consume resources (processing, memory and communications) and add latency overhead. They are, however, different in purpose, and also the nature of the processing involved.

For example, a business rules abstraction layer receives inputs from one or more sources and applies an algorithm which generates an output. An encryption layer (an example of an indirection layer) transforms its input stream into an output stream in a much more straightforward manner, i.e., the actual *content* of the output is the same as the content of the input.

A network router is another example of an indirection layer. It is a store-and-forward device that provides network address translation, may provide firewall services, encryption services (e.g., https) and may also perform a load-balancing function. But, it doesn’t change the actual content of the data being routed.

For each indirection layer that we contemplate adding, we need to ask a set of questions similar to those for abstraction layers:

- Does an indirection layer significantly simplify the process of implementing the solution to the problem being addressed by the application, for example, by enhancing the scalability of the application?

- Does the proposed indirection layer provide any significant benefits? For example, in Deloitte, we separate portions of our web applications into several zones for security reasons.

For example, in the .NET environment, we are encouraged to use ADO.NET and disconnected datasets (an indirection layer), rather than direct connections to the database. ADO.NET utilizes SOAP, which renders the schema of an object in XML, which is rendered in multi-byte Unicode. The schema of the object is obtained by the consumer of the service via a discovery process, and persists for the duration of the object.

Unfortunately, disconnected datasets aren't persistent, so a new dataset object is instantiated remotely each time each time a dataset 'Fill' or 'Update' action is performed via a data adapter, causing the entire schema for the object to be transferred again. This introduces a lot of overhead in the form of the processing necessary to generate the XML, the processing necessary to translate the XML, and the bandwidth necessary to transmit the XML.

One of the benefits of this approach is that disconnected datasets hide the source of the data from the client application, which enhances scalability as well as the ability to host the data using a variety of different products. Another is that a disconnected dataset serves as a local data cache, which can enhance performance considerably, provided that the dataset doesn't need to be updated to the database that often.

One must always ask the question, "Do the benefits outweigh the costs?"

Use of Relational Databases

Microsoft would like to use SQL Server for affinity hiding, simple data storage, etc. But, putting data into a relational database is tremendously expensive. If there is no need to perform relational operations on the data, either now or in the future, alternatives need to be evaluated.

Ideally, we would like to store data in a text file, since such files can be edited by humans directly with a text editor. Where a large amount of binary data is involved, a binary flat file, or even a compressed binary flat file may be better.

Compressed files require more processing in order to compress and un-compress them, but they minimize storage requirements and the time required to send them over the wire.

The flip-side of this is that using a relational database to store 'flat-file' data does provide a lot of the 'plumbing' for free, e.g., authorization and data separation functionality.

Affinity Hiding – Pros and Cons

Microsoft is big on using layered architectures to hide resource affinities from the business logic. This makes designing and scaling the application easier, but it comes at a price – performance. There is a need to evaluate tradeoffs and alternative approaches here.

The fact is, the data always resides somewhere, and the application always executes somewhere. You simply cannot eliminate affinities; you can only hide them from the application.

Whenever you hide an affinity, you necessarily add an indirection layer. This comes at a performance cost, as discussed above. You need some sort of transaction manager in place that knows where the data is physically located, or knows where the application is physically running. It must maintain routing tables that allow the application or database server to reply to requests. Additionally, if there is more than one transaction server, there may be some coordination required between them. All of this increases latency, both due to the processing required in the transaction server and the extra communication paths that must be traversed.

For enterprise applications with a lot of simultaneous users, the performance penalty is worth it in order to allow the application to grow by simply adding more computers. On the other hand, small applications with few users are unlikely to ever benefit significantly from enhanced scalability, and would be better served by a design with better performance.

Economics of Use of Various Types of Interfaces – In-line Code, Function Calls, Intra-process Messaging (threads), Inter-Process Messaging, Cross-Host Messaging, Cross-Site Messaging

The type of interface you choose has a direct impact on performance. Many developers would rather create a function for a few lines of code that they use over again several times rather than using the cut-and-paste function in their editor. But, all of those function calls can really add up, particularly when you are cycling through a section of code 50 times a second.

Spawning multiple threads increases the apparent parallelism of an application, and on computers with multiple CPUs it also increases the actual parallelism. But, does the extra parallelism actually provide any advantages? In some cases, where the user interface must continue to update while the application is performing a long-running task, using separate threads for the user interface and the background task makes a lot of sense. In other cases, the ability to farm out parts of a task to a number of processes running in parallel can greatly increase the processing power that can be applied to solving a problem.

On the other hand, every thread you add to a process adds context switching overhead, adds to the memory requirements, adds to contention for resources such as physical disks, etc. All of this slows down the response time for a single transaction. When multiple transactions are requested simultaneously, it may be more efficient to process them sequentially rather than in parallel. If you have more than one transaction being processed in parallel per CPU, they are really processed sequentially anyway, but the apparent parallelism can add significant overhead without providing any benefit.

Context switching between processes requires even more overhead than context-switching between threads in a single process. So the same rationale applies as with multi-threading.

Cross-host and cross-site interfaces are the most expensive of all in terms of performance. They can add to the security of an application by allowing firewalls to be inserted between layers of the application, e.g., between the presentation layer and the business rules layer, and between the business rules layer and the data layer. They can also be used to enhance scalability when a mechanism is in place that provides affinity hiding.

Regardless of the interface, we must always ask whether the benefits outweigh the cost.

Don't Attempt to Optimize Inter-Host Communications Paths in the Application

Let the network engineers do it. For example, Microsoft has added the ability to specify window size and other parameters in function calls made within the application. Unfortunately, the people who use these calls don't necessarily understand the nature of the links over which the calls will actually be routed. This can result in poor performance over certain types of links, such as satellite links, where specifying the wrong window size can result in a link sitting idle while waiting for an ACK while at the same time keeping it from being used by other applications.

Designing for Operation in Contended Environments

When designing and configuring an application, we need to understand what else is running on the machine(s) that host the application. For example, some other activities that may take place while the application is running are virus scans, disk de-fragmentation, database backups, SMS updates, etc.

These activities can have a significant impact on available CPU bandwidth and memory and network interface resources. It is important to bear this in mind when sizing the application and specifying performance and capacity requirements.

It's also important to schedule these background activities in a way that minimizes the impact to the application.

Sorting and Searching

Many books have been written on these topics. It's important to develop a basic understanding of some of the basic concepts.

For example, when building a table in memory that will subsequently be searched repeatedly using some attribute, one could use an insertion sort during the construction of the table so that subsequent searches of the table could be done using a binary search algorithm.

Factors that Influence Response Time

There are a number of factors that influence the response time of an application to a client request. Here is a non-exhaustive list:

- Effective throughput of the network that connects the client to the application
- Efficiency of the software design and coding
- Size and complexity of the database
- Load on the servers hosting the application
- Power of the servers hosting the application

Some of these factors change as a result of changes that have been made to the application, such as changing the hardware on which the application is hosted, or changing the functionality provided by the software. Others change in a more transitory manner, such as network congestion issues and instantaneous processing load. Still others follow trends over time, such as the size of the database and average load.

Effective Throughput of the Network that Connects the Client to the Application

Effective throughput is an aggregate measure of network performance that incorporates such things as actual bandwidth, latency, congestion effects, jitter and packet loss. It is a directly measurable quantity equal to the amount of payload data that can be sent between the client and the server in a unit of time.

This factor is important to application architecture because of the fact that our applications are globally distributed, with many clients located in remote regions of the world that have poor quality of service and low effective throughput.

When specifying performance requirements, such as the maximum acceptable response time for a particular application, it is important to understand where this response time is to be measured from, and how it might translate to response times seen at other locations.

It is also important to appreciate that in order to optimize response times, every effort should be made to design the application in such a way as to minimize the amount of data being sent between the client and the application, as well as the ‘chattiness’ of the application, i.e., the amount of back-and-forth interaction between the client and the application necessary to perform a task.

For example, in an application that displays an advanced search page, with a series drop-down menus that are dynamically populated based on previous menu selections, there are several design alternatives.

The first, and worst, would be to perform a postback to the server each time that a menu selection is made in order to have the server populate the next menu. This has the undesirable effect that the entire page is re-rendered dynamically and re-sent to the client each time a menu item is selected. When the user is sitting in the same building as the application, the round-trip delays wouldn’t be particularly painful. However, if the user is located in a remote region of the world with poor effective throughput, the round-trip delays are likely to be excruciating.

A better alternative would be the use of Ajax calls. This is a technology that allows only portions of a web page to be retrieved from the server, with the current web page being subsequently updated on the client. There is still a round trip involved, but, in this example, for instance, the amount of information being retrieved from the server is minimal compared to re-downloading the entire page.

A still-better alternative would be to add JavaScript to the web page to dynamically populate the menu. The size of the web page would be larger, because all of the menu alternatives would have to be embedded in the web page that is initially downloaded. However, the postbacks would be eliminated completely.

This last method may not be possible in an application where the drop-down menus are dynamically populated from the actual data being searched, e.g., a set of dynamic filters. So the application designers need to decide if they really want to design their application that way.

The application should also be designed in such a way that it maximizes the percentage of embedded objects that can be cached at the browser, once again reducing the round trips to the server and the amount of data being sent over the wire.

Efficiency of the Software Design and Coding

Response time is equal to the service time for a transaction times the number of transactions in the system. This is one way of stating what is known a 'Little's Law' and it is fundamental to understanding the performance of systems.

Here the service time per transaction is the inverse of the throughput, i.e., transactions per second. In a lightly loaded system with no queuing, service time per transaction is effectively the equivalent of processing time per transaction.

As changes are made to an application over time, and code is re-factored, the processing time that is needed to perform an operation may get better or worse. These changes have a direct effect on response time.

Further, as the load on the system is increased, eventually a point is reach where the application is incapable of handling all requests that it receives simultaneously, and queuing begins to occur. This can have a dramatic effect on response time, since now the response time for an arriving transaction request is equal to the processing time for a transaction times all of the transactions waiting in the queue ahead of the arriving transaction plus the arriving transaction itself.

Not only does increasing the processing time for an operation reduce the load necessary to reach the knee in the performance curve where queuing occurs, but it also increases the slope of the response time curve in the queuing region.

So, when making changes to the application, such as adding functionality, it is important to design in such a way that any negative impact to the processing time of existing functionality is minimized.

Size and Complexity of the Database

Each an every database operation that is performed takes time, and for many of these operations, the amount of data that needs to be processed directly affects the time necessary to perform the operation.

The complexity of the database design also affects performance, since increased complexity increases the size and complexity of the code used to process the data. This can be caused both by design and also by dynamic effects.

For example, searching a database consisting of trees of elements with one or more levels of depth is an example where such a dynamic effect may be observed. Performing a search on such a database, with 1,000 trees with a depth of one, for example, would be expected to complete significantly faster than conducting the same search on a database with 100 trees, each with a depth of ten. The amount of data being processed may be similar, but the number of database operations required to process it is substantially greater.

Typically, when designing an application, these effects aren't readily apparent, and a solution that performs well in a lab with a small database scales poorly when deployed as an enterprise application.

The size, and sometimes the complexity, of databases tend to increase over time, as the user base is increased and as the application is used more frequently. The effects of these increases on response time need to be monitored periodically on an on-going basis for capacity planning and for understanding the effects of potential changes to the application or the user base that are being contemplated.

Load on the Servers Hosting the Application

The effect of load on response time is not straight-forward. Typically, response time for an operation is relatively unaffected by increases in load, up to a point where it is necessary for the application to start queuing requests. This point is known as the knee in the response time curve. Beyond this point, the response time for an operation increases dramatically as the load is increased.

Typically, the arrival rate of transaction requests to an application isn't uniformly distributed. It is typically Poisson-distributed, and the mean rate will vary throughout the day the week, the month, the quarter and the year.

This variation in arrival rates allows queuing systems to be effective. Transactions that back up in the queue during periods of high activity are worked off during subsequent periods of low activity.

However, over time, as the number of users of an application increases, as the size of its database increases, and as the processing time per transaction increases, the frequency with which the application enters a queuing mode increases, and the duration of these queuing mode intervals increases.

It is important to understand this from a capacity planning perspective, since periodic intervals of sharp increases in response times can be monitored, both for their size and duration, in order to predict the future need to add capacity.

Typically, the number of simultaneous users is used as a measure of load. In this case, it's also important to understand the work profile of users, so that changes in load can be properly modeled and analyzed.

Power of the Servers Hosting the Application

The size and speed of the servers hosting the application, including CPU speed, number of processing units, cache, memory, local disk, etc., all affect the performance of an application. In addition, the number of servers used also has an effect.

For example, when going from one web server to multiple web servers, some type of load balancer must be added. This is typically a store-and-forward device that adds a certain amount of parasitic latency to an application. Additionally, consideration for the possibility of such a configuration affects and limits the range of designs that can be used in the application. The number of transactions that can be handled in parallel before queuing occurs is a direct measure of the capacity of the application.

A system administrator once observed that his application was only utilizing 7% of the CPU available, and that he could therefore host the application on a smaller, slower machine without any degradation in performance.

Not exactly. Slower machines do everything slower, and such a move would have a direct impact on response times, since the processing time for each and every operation performed would be increased.

Place Data as Close as Practical to Where it is Used

This includes putting databases on the same host machines as applications where practical, co-locating applications and databases on the same LAN, and providing high-speed interfaces between applications and databases. The point is to minimize latency and maximize bandwidth between applications and the data that they use. It should be noted that security requirements can impose significant constraints on the ability to do this.

Placing data as close as possible to where it will be used also means the judicious use of caching. Where data is static or infrequently changing, an application may cache the data in an object that is globally accessible to all sessions utilizing the application. This can be done, for example, through the use of disconnected datasets that get updated periodically but infrequently.

Make Use of Client-Side Validation of Inputs

Wherever possible, it is important to validate inputs on the client side, before submitting a request to the server. Of course, the server still needs to validate inputs as well, but by validating the inputs client-side first, many useless round-trips can be eliminated.

Move Data Only When Necessary

Don't ever move a block of data when passing a pointer will do. Likewise, don't ever move data more than once when it is possible to move it only once and then reference it again by passing a pointer.

Minimize the Amount of Data Retrieved From the Database

Many times, applications are designed using "SELECT * WHERE ..." statements to retrieve data from the database for intermediate transaction processing steps, such as selecting a task from a task list, or displaying a list of records available to a particular user. Sometimes, these database requests return far more data than the developer had originally anticipated.

When this happens in a part of the application that is executed frequently, such as an intermediate step in processing frequently used forms, the resulting flood of data can have a serious negative impact on the performance of the application by overloading the database server and flooding the back end communications.

Care must be taken to make the 'WHERE' clause as restrictive as possible.

Use Disconnected Datasets with Care

Microsoft ADO.NET, for example, provides the ability to use a “disconnected dataset.” This is a locally cached copy of some set of data that is used by an application. The use of this approach has a number of implecations.

It is expensive to open and close a database connection. This is what happens each time you load or update a disconnected dataset. On the other hand, when you have a large number of simultaneous users accessing the same data, it is a bad idea to lock the tables or rows that they are accessing while the rows or tables are being worked on, since this tends to serialize the users, i.e., their operations are performed sequentially, one user after another, while all of the other users wait.

For a significant number of types of data, there are many consumers but only one producer. For example, the record of a sale is entered one time by the cash register where the sale is recorded, but may be accessed many times by different business applications, such as accounting programs, inventory control programs, etc.

You should never leave a database connection open when you are doing long, complex operations between data accesses. For a single user application, this will result in improved performance. However, for a multi-user application, this will degrade performance significantly.

In addition, when a dataset is used, the entire schema is passed over the wire in XML Unicode text format. Schemas are only supposed to be passed once for the duration of the object. But in the case of datasets, the object only exists for the duration of the fill or update call. Outputting the dataset in binary doesn’t work, because .NET still adds the schema in XML Unicode text format, resulting in even more data being transferred.

Where Possible, Use Unmanaged Code

For example, Microsoft provides the .NET framework. This framework supports “managed code” which is code that runs in an environment where dynamic memory use is managed by the framework, not the application.

Managed code is easier to write because the garbage-collection on the managed heap means we don’t have to de-allocate memory after we use it. Unfortunately, the cost of this ease of coding is performance. Simply put, unmanaged code executes faster.

In C#, unmanaged code is created by marking unmanaged code blocks with the “unsafe” keyword and compiling with the “/unsafe” compiler switch. C++ code is unmanaged by default.

In C++, in order to place an object on the ‘managed heap’, we must first “box” it by adding the ‘^’ character after the object type in the declaration. For example, in C++:

```
String ^ s1 = “Managed string object.” ;
String s2 = “Unmanaged string object.” ;
```

Objects, such as strings, are passed from sections of code written in C++ to sections of code written in C# by placing them on the managed heap. Therefore, such objects must be boxed.

Profile Your Code

In order to maximize performance, it helps to focus on the areas of your code that use the most time, either because they are called the most frequently, or because they take the most time to execute, or both. Profiling can be performed with one of the many profiling tools available on the market, provided you can convince your management to purchase one. Failing that, you can create your own.

You can create your own profiler by adding conditional code to your software that sets a timer at the beginning of the code of interest, and logs the time in the code at the end of the code of interest.

Use the Data Adapter ‘UpdateBatchSize’ Parameter

In ADO.NET, when using disconnected datasets, if you don’t use this parameter, then when you perform an update, the data adapter will make a complete round-trip to the database server for each row that is inserted, updated or deleted. (Note: The SQL Profiler won’t show any difference, but the network round-trips will be saved.)

When doing this, you can use the ‘DataSet.GetChanges’ method or the ‘DataTable.GetChanges’ method to create a new dataset with only the changes that need to be applied. With the ‘DataSet.GetChanges’ , you may get rows that haven’t actually changed, but are needed to satisfy the current existing relations in the dataset. This dataset is then used to update the data source. After doing the update, you would then use the ‘DataSet.AcceptChanges’ method to set the current row values to be the original values and mark all rows as ‘unchanged’ in the original dataset. For large datasets, this minimizes the amount of data that you would pass over the wire.

As the size and complexity of the dataset increases, the amount of work required to perform a ‘GetChanges’ operation on an entire dataset can increase exponentially, so at some point it makes sense to perform the ‘GetChanges’ operation on individual tables instead.

Disable ‘Viewstate’ Where It Isn’t Needed

When using Microsoft Web Forms, Viewstate is data that is used to populate the controls in the forms. Only controls contained within a ‘`<form runat=server>`’ tag in the ‘.aspx’ page can store viewstate. It is stored as key-value pairs using the ‘`System.Web.UI.StateBag`’ object. When the page is posted back to the server, the viewstate data is sent to the server where .NET uses it to construct the state of the controls on the page during initialization. In other words, every time a page posts back to the server, the viewstate data makes another round trip. If there is a lot of it, there can be a significant performance impact, due both to latency and bandwidth issues, and also to the processing necessary at the server in order to deal with it.

Viewstate is turned on for every control on every page by default. It makes sense to disable it for controls where it isn’t needed.

For the following controls, the amount of viewstate data required is minimal, so they can be ignored: ‘HyperLink’, RadioButton’, ‘CheckBox’, ‘TextBox’, ‘Label’, ‘Button’, ‘LinkButton’ and ‘ImageButton’.

For these other controls, the amount of viewstate data can be considerable, so they should be examined carefully: ‘ListBox’, ‘DataGrid’, ‘DataList’, ‘DropDownList’, ‘CheckBoxList’, ‘RadioButtonList’ and ‘Repeater’.

On the other hand, session data can be stored in viewstate rather than a database. This may make sense when using a web server farm and where there isn’t very much session data, it isn’t sensitive in nature and it doesn’t time out. That way, you save round trips to the database from the web server to retrieve session state data each time a page is posted back. You use the ‘Page.IsPostback’ property in the ‘Page_Load’ event of the page, and only go to the database the first time that the page is loaded. Determining when this strategy makes sense is an empirical matter and differs for each application.

Wherever Possible, Avoid Using the ‘finalize()’ Method in Managed Code

I .NET, the ‘finalize()’ method allows extra code to be executed for cleanup when an object is de-referenced. Unfortunately, this code is executed by a special thread that is activated when the pointer to the object is moved from the Finalize Queue to the Freachable Queue during the garbage collection process, and the garbage collection process occurs at unpredictable times. If the ‘finalize()’ code is used to free resources, they don’t actually get freed until some time well after the object is no longer in use.

Additionally, if the ‘finalize()’ code relies on the existence of other objects that may also have been de-referenced, there is no way of predicting which object the garbage collector will process first, leading to un-predictable results.

And lastly, from a performance perspective, because of the way that garbage collection in .NET works, the ‘finalize()’ method has the effect of causing two garbage collections to be required in order to completely remove the object.