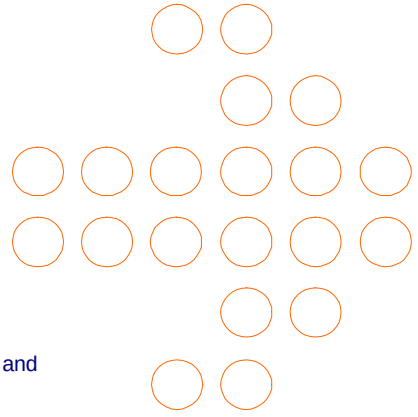
REUTERS

Optimizing Data Retrieval in MIX

Reuters Market Information Express



Prepared by: Jim Foley
Date: July 2004, July 2005

REUTERS

SCOPE AND TARGET AUDIENCE

This document describes a number of approaches to improving MIX server performance by minimizing data retrieval time. Topics include pipelining, cache duration, preloading, and prefetching. A summary is provided.

Throughout this paper, the term "MIX" is used for the product line sometimes called "BIT", and the terms "BBC" and "RLSR" are used more or less interchangeably.

This is a technical document for Reuters solutions developers, application developers, and technical relationship managers. Client technical staff may find this information useful in decision-making or configuration.

Related information may be found in the whitepapers "Response and Throughput Standards", "Caching Overview", and "Caching Views and Analytics in MIX" as well as the MIX and RLSR installation manuals and the MIX Developer's Guide. Readers interested in how "streaming" data is cached should consult the whitepaper "The RMIX Electrified Datastream".

CONTENTS

Introduction

Generally speaking, MIX objects are about data retrieval. Their job is to retrieve and display such things as market data, headlines, and so on, from a remote data service. When we think about this in terms of performance, it's clear that if we can speed up the retrievals, we can speed up the display time of any given web page. What may be less obvious is how significantly this can affect the overall performance of the MIX server.

To fully appreciate the benefit, consider the life of the common web server. A limited pool of threads – typically twenty – is allocated to handling page requests. As each incoming request is filled, its thread is returned to the pool, and from there it is allocated to another request. We can imagine that there might sometimes be hundreds of requests queued up, but only twenty can be processed at a time. This is true even if the system is experiencing idle processor time, which leads to the question: why only twenty threads? Why not a hundred? The answer, of course, is that each additional thread creates additional overhead and additional liability. Having too many threads will mean the system sometimes gets overloaded.

Rather than increasing the number of available threads, it's better to smooth out the workload by designing pages that have as little idle time as possible. If each page finishes quickly, threads return to the pool quickly and the server can handle more requests. This paper describes several techniques that let MIX pages spend less time waiting for data, leading to both a faster page experience and better overall performance of the MIX server.

Pipelining

In MIX, "pipelining" refers to a technique of overlapping data retrievals as a way to minimize the effects of network latency. It works only with static-mode requests, and only with objects that have been enabled for pipelining, but it can cut execution time by half or more.

Figure 1 shows the timing of a non-pipelined MIX request and response. The Retrieve() method sends a data request upstream, then waits for the response before returning control.

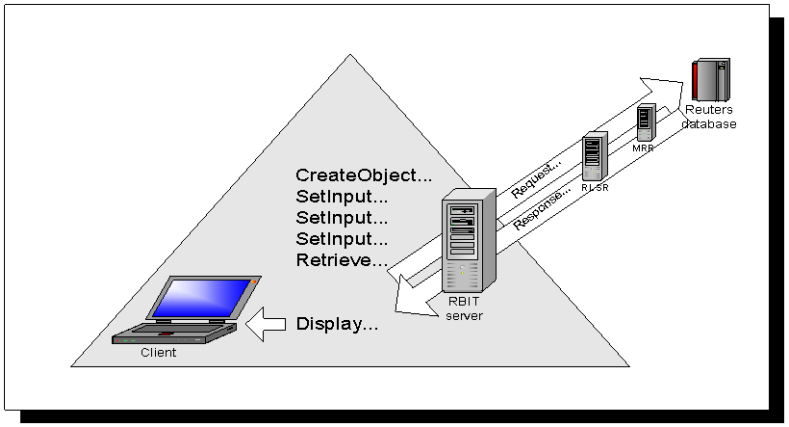


Figure 1 – Un-optimized method. The Retrieve() method blocks as long as the request and response are in the pipeline (the area outside the triangle).

We say that the Retrieve() method *blocks*, or halts execution of the page, during the time between sending the request and receiving the response. The thread is idle during this latency period.

In MIX 3.x, many objects are now "pipeline-enabled", meaning that they do not block on the Retrieve() call. Instead, they simply send the retrieval request upstream and immediately return control. As long as there is no attempt to display or otherwise process the requested data (which we know is still upstream, "in the pipeline"), the thread can continue executing the page. No blocking will occur until an extraction call is issued – i.e., Display(), Value(), and so on. (Of course, these methods must have data to work with, so if the data hasn't arrived in the local cache yet, they have to wait for it).

With pipeline-enabled objects, we leverage this trait of non-blocking retrieval by issuing a series of retrievals in rapid succession. This puts multiple data requests into the pipeline at the same time. When we subsequently issue a series of Display() calls for these retrievals, only the first one is likely to have to wait for data, because all the responses will come back out of the pipeline at about the same time. Latency is not eliminated, but it is reduced because all the latencies transpire concurrently. It's the difference between retrievals occurring one after the other, versus all at the same time. When we speak of "pipelining", we are generally referring to this technique of multiple, simultaneous retrievals.

Using pipelining

The following two code excerpts demonstrate the technique. Assuming an average network latency of ¼ second for a given site, the first example would take slightly over ½ second to execute, compared to just over ¼ second for the second example.

```
// This code is relatively inefficient:

// InstrumentFundamental
InstF.SetInput(...);
InstF.Retrieve();        // request goes into the pipe
InstF.Display(...);      // wait ¼ second for response

// NewsSearch
News.SetInput(...);
News.Retrieve();      // request goes into the pipe
News.Display(...);    // wait ¼ second for response
```

```
// This code is more efficient:

News.SetInput(...);
InstF.SetInput(...);

// retrieve
InstF.Retrieve();        // request goes into the pipe
News.Retrieve();      // request goes into the pipe
```

```
// display
InstF.Display(...);      // wait ¼ second for response
News.Display(...);     // probably will not have to wait
```

Sequencing on retrieval time

This tremendous gain in server capacity can be tweaked a little further, if we remember that not all retrievals are equal. Performing a static news search, for instance, typically takes longer than retrieving a specific news story by its story ID. So if we're doing both in the same page, we take advantage of the different retrieval times by starting the slowest retrieval first.

```
// retrieve
NewsSearchObj.Retrieve();          // slowest retrieval goes into the pipe first
NewsStoryObj.Retrieve();           // fastest retrieval goes into the pipe last

// display
NewsStoryObj.Display(…);           // ideally, fastest retrieval is displayed first
NewsSearchObj.Display(…);          // and slowest retrieval is displayed last
```

A more complete sample script, included at the end of this document, demonstrates pipelining technique and may help estimate the potential time savings.

Checking a retrieval status

Since the only way to check the status of a Retrieve() is to wait for the data to arrive, in pipeline-enabled objects the Status() and StatusString() methods block if a retrieval is pending. (In fact, nearly all the helper functions will block in a pipeline-enabled object.) It's important to be aware of this. Following a retrieval with an immediate status check causes the object to behave as it did before it was pipeline-enabled. The answer in most cases will be to check the retrieval status just prior to issuing a Display(), Value(), or other extraction call.

MIX support for pipelining

As of MIX version 3.6.5, a dozen objects are "pipelined". The following table lists the objects and shows which version introduced pipelining support for them.

Table 1: Objects that support standard pipelining.

Prior to 3.6.0	Version 3.6.0	Version 3.6.5
InstrumentFundamental	LookupByAlias	CompanyList
NewsSearch	Collection	LookupBySymbol
NewsStory	RapidProxy	LookupByName
NewsCategories		
NewsSources		

Note that the Instrument object is not on the list of pipelined objects. Static Instrument requests do not follow the request/response model – that is, not in the normal sense of a request going to the central system and a response coming back. Instead, a Retrieve() for any instrument data (assuming it is not found in the cache) simply subscribes the local cache to continuing updates for that instrument, and all requests are then served from the cache. The Retrieve() blocks until the data begins to arrive, at which point it grabs what it needs from the cache and returns. The cache continues to receive updates for a configurable period of time, during which the latest data on that vehicle is available to any local client who requests it. This approach blurs the usual distinction between static and dynamic data, and in practice means a high percentage of Instrument requests are serviced locally from cache. For this and other reasons, pipelining as described in this paper is not enabled for the Instrument object.

Managing the cache

The caching system is described in considerable detail in the paper "Caching Overview", but worth a quick mention here. In general, we can say that the RLSR and the MIX attempt to service data requests from cache whenever possible. A "cache hit" is much faster than a roundtrip to the central system to retrieve data. With this in mind, there are several ways of optimizing the use of the cache to reduce idle time.

Setting the cache duration

For static data requests, caching involves holding the results of a request for some period of time, in case an identical request appears within that period. For dynamic data requests, caching involves maintaining a subscription for some period of time after the last client has lost interest, in case another client wants data from the same subscription. The period of time in both cases, referred to as the "cache duration", is configurable. We can also configure the size of the cache, so that it can hold more or fewer items. By increasing the cache size and duration, we increase the likelihood of a given item being found in the cache. This results in faster responses and fewer upstream requests.

Of course, this speed and efficiency comes at a cost. With a static item, such as a news story or an analytic chart, the faster answer contains slightly-older information. With a dynamic item, such as a streaming series of trades, we're committing some bandwidth to receiving updates for an item long after anyone's actually interested in it. (Of course, with a low-interest item, updates are often fairly infrequent.) And with both static and streaming data, we're using cache space to store data that may never be requested.

There's another caveat, one that involves data authorization. Every data item must be authorized, i.e. approved for use by that client. (See "Authorization in RDN" and "Authorization in RAPID" for more on this.) With subscription data, this takes place at the MIX server, but with request/response transactions, it typically happens upstream at the central database. If a cache hit is intended to avoid the upstream trip, how do we deal with authorization? It gets a little tricky here. The system assumes that an identical request, made with the same user number or an identical authorization profile, would return exactly the same data within the

cache duration period. So it caches the information it needs to leverage this assumption. The MIX server caches the user number with the data, and the RLSR caches the profile with the data. Subsequent identical requests, if they have an identical user number or profile, can thus be served from the cache, assuming that they occur within the cache duration period and the item hasn't been purged due to cache size limits.

What are the chances that a request, say for a popular news story, will occur repeatedly with the same user number? Very high, if the site uses auto-login, a feature that automatically assigns every user the same ID and authorization profile. (See "Packages and User Login" in the MIX Developer's Guide for more on this.)

Sites that prefer not to use auto-login can achieve similar caching efficiencies by using simple permissions structures (a "role" approach) – for instance, three levels of users, based on three authorization profiles. With this approach, on the MIX servers the cache duration for static items is set to zero, forcing static requests up to the RLSR cache. There we might expect the cache to contain three copies of the popular news story, each associated with one of the three profiles. On such a site, no matter how frequently users request the story, the RLSR will request a fresh copy from upstream only after the cache duration runs out – say, every five minutes.

Global cache duration

Cache duration and size are set in configuration files on the RLSR, BBC, and MIX server. There, each object has its own settings for cache duration, with separate settings for dynamic data. Details are found in the paper "Caching Views and Analytics in MIX", the "Caching Overview" paper, and the relevant installation guides.

Objects that support the Cache_Duration field:

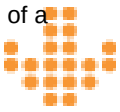
Analytic
Collection
CompanyList
InstrumentFundamental
InstrumentInterdayHistory
InstrumentIntradayHistory
InstrumentTickHistory
LookupByAlias
LookupByName
LookupBySymbol
NewsCategories
NewsSearch
NewsSources
NewsStory

Cache durations in individual object instances

A number of MIX objects support a ".Cache_Duration" field – generally, the objects that request data that can be cached. This allows the solution developer to specify a cache duration for an individual data request. As suggested in the "Caching Overview" paper, this might be an effective approach for caching a main page that displays the same data repeatedly to all users – e.g., a static list of topical headlines. The specified cache duration would apply only to this exact headline request.

Preloading the cache

In some situations, a performance gain in Instrument retrievals can be realized by "preloading" the cache. The idea is simple: if a particular vehicle is likely to be requested, subscribe to it before the request is actually issued. The difference in retrieval time is quite dramatic. Pulling existing instrument data from the MIX server's cache typically takes one or two hundredths of a



second. By comparison, starting a new subscription and waiting for the data may take half a second or more.

The preload list

On the MIX server, preloading uses a manually maintained list of symbols for the most popular instruments and collections. Some preload lists may contain thousands of symbols. The actual preloading begins when any client loads a page from within the virtual directory associated with the package (e.g., www.example.com/package/), and the process is staggered so as not to overwhelm the server or channel. Then, at regular intervals, subscriptions to all the listed symbols are updated so that they don't expire.

The value of list-based preloading is limited. Once a vehicle has been requested – say, during the morning's opening bell rush – it has a good chance of being served from cache all day, so only the first request for an item shows improvement from preloading. The preload list may be most useful at a small site without an RLSR, where it can reduce the strain caused by large numbers of simultaneous new subscriptions during the morning rush.

Preloading a list on the MIX server is gradually being deprecated. Instead, we suggest using a long cache duration on the RLSR, to maintain ongoing subscriptions to popular vehicles at that level, and setting short cache durations on the MIX servers. Because of the large user pool of an entire server farm, cache hits are very likely on the RLSR, so this gives the MIX server rapid data access without high caching overhead.

For more on the preload list, see the "Packages and User Login" section of the MIX Developer's Guide, plus the various relevant installation guides. For more on how an RLSR caches vehicles, see the Instrument chapter of the MIX Developer's Guide.

Preloading while executing a page

Preloading can also be implemented on the fly when using objects that return a list of instrument symbols. A user who requests such a list is likely to want details on at least one of the listed vehicles. So these objects support a feature called "preloading", which is enabled by setting the object's ".Preload" field to "True". When preloading is enabled, the MIX server automatically subscribes to any vehicle returned by the search, so a subsequent request for this instrument can be serviced very quickly from the cache. (Or, at least, the vehicle's most popular fields, as explained in the section on "Prefetching fields".)

Objects that support preloading:

Collection
CompanyList
LookupByAlias
LookupByName
LookupBySymbol
Ranking

This type of preloading does not add the vehicle to the server's preload list.

Prefetching fields

At the time of this writing, eight MIX objects are enabled for "prefetching". These objects are designed to retrieve from the central system only a subset of their available fields, and they

offer a ".Prefetchlist" field in which additional fields can be specified for retrieval. Such objects automatically conserve bandwidth by reducing the amount of data transmitted, but if improperly used, they can create idle time by causing additional trips upstream for data. We'll look at the prefetch concept in three contexts, as follows.

Large fieldtrees

The Instrument object has hundreds of fields. Clearly it would be a waste to transmit all of these fields when most applications need only a few of them. For this reason, the Instrument object by default retrieves only a standard set of the most commonly-used fields. If an application attempts to reference one of the unretrieved fields – with Value(), Display(), or most other helper methods – the MIX server has to send upstream for the missing fields. On the other hand, if all the needed fields are declared in ".Prefetchlist" in advance of the Retrieve() call, they can all be acquired in a single trip. This results in less trips and less idle time, improving page speed and overall server performance.

Historically we have not documented the list of fields that are included in the "standard set", as this may be subject to changes which could dramatically affect performance. The suggested best practice is to list in the ".Prefetchlist" field all the Instrument fields that will be displayed in the page – and, of course, avoid listing any fields that aren't actually used.

Once an additional field is added to a subscription – via a display method or the ".Prefetchlist" field – the server will continue to receive data for that field until the entire subscription expires.

High volume

In its dynamic mode, NewsSearch works much like Instrument: the MIX server streams headlines to a large number of clients from a single subscription that brings in the entire Reuters universe of headlines. (Static headline requests, in contrast, go to the central database for request/response service.) This creates a lot of wire volume. To save bandwidth, the Symbol field and several category fields are by default not included in the subscription. These are not large fields, but because the volume is so high, the savings is significant. If these fields are to be used in a page, they should be specified in the ".Prefetchlist" field – otherwise, an additional round trip will be made every time one of them is displayed.

A similar implementation of ".Prefetchlist" is being considered for the new InstrumentTickSeries object, which does not have a large fieldtree but does have more fields than are needed by most applications, and like Instrument and NewsSearch is a subscription-based, high-volume object oriented to high performance.

Large fields

Other objects that support prefetching have relatively small fieldtrees that include one or two large fields. This is the case with some simple "lookup" objects – LookupByAlias, LookupByName, LookupBySymbol, and CompanyList. With these objects, bandwidth is conserved by not retrieving the lengthy description field unless it is needed.

A more dramatic example of this principle is found in the NewsStory object. A retrieval of NewsStory returns peripheral information such as date and time, news category, and so on. But by default, it does not return the actual body of the story. Since the story is usually the largest part of the fieldtree, this saves a lot of bandwidth. If the body is to be displayed or otherwise referenced, it should be specified in the ".Prefetchlist" field to avoid causing an extra round trip.

It should be clear to the reader that fields which can be mentioned in ".Prefetchlist" should always be listed there if they are to be referenced, to save the extra round trip. Conversely, fields that will not be used should not be prefetched.

Other notes

The UConvert object, which converts a string's encoding format, uses a ".Prefetchlist" field to specify the output format. This is essentially an API-consistent way to set up the object, and is unrelated to the primary use of ".Prefetchlist" discussed here. In fact, UConvert never goes upstream for information, but simply performs conversions locally on the MIX server.

The FixedIncomeCalculation and FixedIncomeSearch objects, which have even more possible fields than Instrument, don't have a standard set of fields that are returned by default, so they don't use a ".Prefetchlist" field. Instead, these objects use an "optional fields" model, in which the application must specify *every* desired data field before calling Retrieve(). There's no trick for improving on this automatic efficiency. The "optional fields" feature is only mentioned here for clarity and completeness.

More information on using ".Prefetchlist" and optional fields be found in the relevant object chapters of the MIX Developer's Guide. Note that the prefetch features are unlikely to be supported with other namespaces – e.g. ".Prefetchlist" is not supported with Instrument when using the RAPID namespace, or with NewsSearch when using the NDS namespace.

Summary

- A page processing thread has idle time while waiting for a data request to come back from the network. When all threads are idle at the same time, the server is idle – even if it has other client requests waiting to be assigned a thread. If idle time is reduced, the user has a faster experience and the server supports more users.
- Always use pipelining technique in objects that support it. Generally, this means issuing all the Retrieve() calls before issuing any display calls, and calling the slowest retrieval first.
- Increasing a site's cache duration may increase the cache hit rate – especially with auto-login packages, where everyone has the same user number. With request/response data, this results in faster response, but slightly older data. With subscription data, this keeps more subscriptions immediately available, at some cost in space and bandwidth.

- Similarly, in a frequently-accessed page that displays popular data, such as the main page of a public site, it may be good to increase the cache duration for the individual object instance.
- At sites that do not use an RLSR, use preloading at the package level to smooth out the server response during new-subscription rushes, e.g., at the morning bell. As with an increase in cache duration, remember that this will increase a site's use of cache and bandwidth.
- In objects that support prefetching, pay close attention to the fields used, and declare them as appropriate in the ".Prefetchlist" field before issuing any Retrieve() calls.

Sample scripts

Pipelining example

Below is a sample ASP script that compares non-pipelined versus pipelined requests for an array of instances of the InstrumentFundamental object. The difference between the two average retrieval times gives an approximation of the site's average latency during the time the script was executing. Actual results will vary but the performance effect of the pipelining technique should be clear. With that in mind, the script output should look something like this:

```
Retrieving... 50 symbols.
Average pipelined time: 0.011 seconds.
Average non-pipelined time: 0.059 seconds.
```

Script 1: Comparison of pipelined and non-pipelined requests for InstrumentFundamental.

```
<%@ Language = JScript  %><%
function CurrentTime() {
      var t=new Date();
      return(t.valueOf());
}

var UserObj = Server.CreateObject("Bridge.User.3");
var LookupObj = UserObj.CreateObject( "Bridge.LookupByName.3" );
var FundObj = new Array();
var TestSize = 50;                        // number of test instances

// Get an arbitrary list of symbols from LookupByName
Response.Write("Retrieving... ");
Response.Flush();
LookupObj.SetInput(".Name",  "a");
LookupObj.SetInput(".Max_Count", TestSize);
LookupObj.Retrieve();
```

```
TestSize = LookupObj.InstanceCount(".Lookupresult");
Response.Write(TestSize + " symbols.");
Response.Flush();

// Give them to the InstrumentFundamental objects for retrieval
for (var i=0; i<TestSize; i++) {
      FundObj[i] = UserObj.CreateObject("Bridge.InstrumentFundamental.3");
      FundObj[i].SetInput(".Instrument", LookupObj.Value(".Lookupresult[" + i + "].Symbol"));
}

// "PIPELINED" (CONCURRENT) RETRIEVAL / DISPLAY
StartTime = CurrentTime();
for (var i=0; i< TestSize; i++) {
      FundObj[i].Retrieve();                              // ***** RETRIEVE
}
for (var i=0; i< TestSize; i++) {
      var a = FundObj[i].Display(".Instrument");          // ***** DISPLAY
}
Response.Write("<br>Average pipelined time: ");
Response.Write(Math.round((CurrentTime()-StartTime)/TestSize)/1000 + " seconds." );
Response.Flush();

//  "NON-PIPELINED" (CYCLICAL) RETRIEVAL / DISPLAY
StartTime=CurrentTime();
for (var i=0; i< TestSize; i++) {
      FundObj[i].Retrieve();                              // ***** RETRIEVE
      var a = FundObj[i].Display(".Instrument");          // ***** DISPLAY
}
Response.Write("<br>Average non-pipelined time: ");
Response.Write(Math.round((CurrentTime()-StartTime)/TestSize)/1000 + " seconds." );
Response.Flush();

%>
```

Instrument subscription time

The following sample script can be used to estimate the time required to retrieve data for an "unmarked" vehicle – that is, a vehicle that is not already in the cache. It requests data for some vehicles that are unlikely to be found in the cache, then requests the same data again. The times for the two retrievals should be very different, because the first retrieval has to go upstream for the data, and the second retrieval doesn't.

Note that this is not a demonstration of the time required for a normal request/response "round trip" from the MIX server to the central system, because Instrument data does not use the request/response pattern. Rather, the first retrieval shows the time required to subscribe to a vehicle and receive the first update, and the second retrieval shows the time required to fetch the latest data from the cache. Actual results will vary from one site to another, but will likely look something like this:

Retrieving... 20 symbols.
1st run, average retrieval time: 0.327 seconds.
2nd run, average retrieval time: 0.01 seconds.

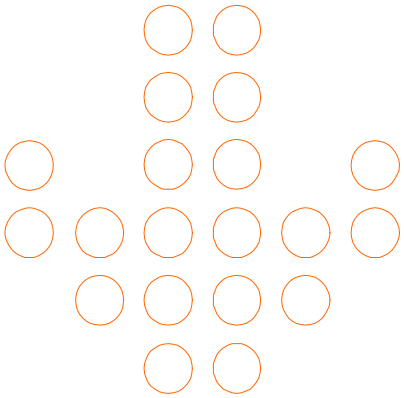
Obviously, once the script has been run, the requested vehicles will remain in the cache for a while, so subsequent tests will give the same results for the 1st and 2nd runs. For this reason, several extra sets of symbols are provided as comments in the script, so that the test may be run several times in a brief period.

It's interesting to note that uncommenting the Display line in the RetrievalTime function of this script makes no significant difference in performance. This confirms that the Instrument object blocks at Retrieve(), not at Display().

Script 2: Getting a sense of the subscription time for instrument data.

```
<%@ Language = JScript %><%

function CurrentTime() {
	var t=new Date();
	return(t.valueOf());
}

function RetrievalTime() {
	StartTime = CurrentTime();
	for (var i=0; i<TestSize; i++) {
		InstrObj.Retrieve();
		//Response.Write(InstrObj.Display("Instrument[" + i + "]") + "<br>");
	}
	return Math.round((CurrentTime()-StartTime)/TestSize)/1000;
}

var UserObj = Server.CreateObject("Bridge.User.3");
var InstrObj = UserObj.CreateObject("Bridge.Instrument.3");

var TestSize = 7; // Number of vehicles in sample
var VehicleStr;   // Uncomment a different string below if running more than once.
VehicleStr = "us;AAMC|us;APRDO|us;ATXAN|us;AVOC|us;CBYAA|us;CCIG|us;CEDNM";
//VehicleStr = "us;CILTL|us;COFF|us;CUMD|us;CYSM|us;DSR|us;EGYKO|us;EYGUO";
//VehicleStr = "us;EYLAP|us;EYMSN|us;FBAK|us;FHCS|us;GOKN|us;GSU,B|us;IFST";
//VehicleStr = "us;MEWX|us;MNESP|us;MPRWL|us;MWPSL|us;NAPE|us;NMS|us;PFIT";
//VehicleStr = "us;PUSH|us;RDGA|us;SPTM|us;THLM|us;UFBC|us;WREGO|us;WSPBP";
var Vehicles = VehicleStr.split("|");
for (var i=0; i<TestSize; i++) {
	InstrObj.SetInput("Instrument[" + i + "]", Vehicles[i]);
}

var Average1=RetrievalTime();
Response.Write("1st run, average retrieval time:  " + Average1 + " seconds.<br>");
var Average2=RetrievalTime();
```

```
Response.Write("2nd run, average retrieval time: " + Average2 + " seconds.<br>");

if (Average1/Average2 < 1.2 ) {        // if the difference is not significant...
      Response.Write("<br>Try a different set of vehicles.");
}

%>
```



REUTERS