Scaling .NET Web Applications with Microsoft’s Project Code-named “Velocity”

A Technical White Paper by Grid Dynamics  
September 1, 2009

Maxim Martynov  
Sean Kumar  
Hal Jespersen

|  |
| --- |
| **Abstract:** Grid Dynamics created three sample applications, designed to be typical use cases of Microsoft’s highly scalable in-memory cache―the project code-named “Velocity”―and ran extensive benchmarking tests to evaluate their performance characteristics. The applications were a blogging engine, demonstrating the most basic features of Velocity caching technology; a simple e-commerce website, demonstrating Velocity’s capabilities for managing session state, and; a market data application, demonstrating Velocity’s event processing capabilities. Building these applications with the Velocity project’s in-memory application cache platform achieved superior, linear scalability when compared to the same applications built without Velocity. This paper presents the detailed results of these experiments, describes how the features of Velocity can be used in a real world application, and also offers a comparative evaluation of the important features of a few application-scaling middleware packages, including Velocity. |

*Scaling .NET Web Applications with Microsoft’s Project Code-named Velocity*

© 2009 Grid Dynamics Consulting Services, Inc. – All Rights Reserved

Grid Dynamics

39300 Civic Center Drive, Suite 145

Fremont, CA 94538

Tel: +1 510-574-0872

Fax: +1 636-773-4528

<http://www.griddynamics.com/>

Table of Contents

[Overview of Velocity 5](#_Toc235491653)

[Application Benchmarks 6](#_Toc235491654)

[Velocity as a Distributed Cache 6](#_Toc235491655)

[Velocity Session State Provider 20](#_Toc235491656)

[Velocity Notifications 31](#_Toc235491657)

[Creating a Real-World Application 37](#_Toc235491658)

[Feature Comparisons 39](#_Toc235491659)

[Conclusions and Recommendations 42](#_Toc235491660)

[Appendix A: Blog Engine Additional Details 43](#_Toc235491661)

[Appendix B: E-Commerce Website Details 52](#_Toc235491662)

[Appendix C: Market Data Application Details 60](#_Toc235491663)

[About Grid Dynamics 61](#_Toc235491664)

# Overview of Velocity

Microsoft’s project code-named “Velocity” provides a highly scalable in-memory cache for application data. By using a cache, you can significantly improve application performance by avoiding unnecessary calls to the data source. A distributed cache enables your application to match increasing demand with increasing throughput by using a cache cluster that automatically manages the complexities of load balancing. Scalability can be achieved by adding more computers on demand. Higher availability can be achieved by allowing Velocity to store copies of the data across the cluster. Velocity can run along with any distributed application on the same server or it can be accessed over the network on a remote server.

Velocity is currently distributed as a Community Technology Preview (CTP). The version of Velocity evaluated in this white paper is known as CTP3, which has been available since April 2009. A new version, CTP4, is expected to be available for downloading late in the fall of 2009.

# Application Benchmarks

Grid Dynamics implemented sample applications that represent typical Velocity use cases. These applications demonstrate how Resource, Activity, and Reference data can be cached in different ways and for different purposes. They were not fully functional applications with complete user interfaces, but were designed to illustrate typical cache usage patterns and to produce reliable, instrumented results.

The three sample applications developed were:

* Blog engine application, which uses Velocity as a Distributed Cache
* E-Commerce website, which uses the Velocity Session State Provider
* Market-data application, which uses Velocity Notifications

Using these applications, Grid Dynamics performed benchmark tests to capture several metrics: application-specific metrics such as throughput and latency, and system metrics such as CPU load, RAM usage, and network usage.

## Velocity as a Distributed Cache

A blogging engine is a good example of a web application that is data intensive. Each page displays data such as friend lists and friend feeds, which are typically lists of items that require complex database queries to fetch. Also, this information is displayed in many web pages on the site, which means the same data is accessed many times. A blog engine can get a huge performance boost by utilizing a distributed cache.

To demonstrate the performance gains that could be achieved by Velocity we created two versions of the same application; the first version used only a Database and the second version used Velocity to cache the data retrieved from the database.

The application simulated users that can write posts, leave comments to the posts, and can add other users to a friend list (including a friend’s posts in the user friend’s feed)

### Architecture

The architecture of the blog engine application is typical of a web application and can be summarized as follows:

* A database stores application data, containing tables with users, relationships between users, blog posts, and comments to blog posts.
* Horizontally scalable web servers perform the basic functions of creating a blog post, viewing a blog post, editing a post, adding a user as friend, requesting a friend feed, and adding a comment to a post.
* (non-Velocity case) Each request to the web server generates a request to the database followed by some additional processing of raw data and applying domain logic.
* (Velocity case) Each request to the web server generates a request to Velocity and, in the case of a cache miss, to a database, followed by some additional processing of raw data and applying domain logic. The request to Velocity does not result in additional processing. Velocity contains caches for posts, users, friend lists for the user, and a friend feed for the user.



Figure 1: Blog Engine Application Architecture

The database used for the tests was SQL Server. The specific hardware and database configuration used in the tests are shown in Appendix A.

### Velocity usage

Velocity was optimized for read-write access and data was pre-cached in Velocity. The following objects were stored in the cache:

* For each user a list of the top 20 post IDs from the user’s blog
* For each user a list of the top 20 post IDs from the user’s friend feed
* For each user a list of user IDs of friends
* Each post with comment count
* For each post a list of the top 20 comments

The application used the cache for the following operations:

* Each blog or friend feed view operation reads the index for the user from the cache and then retrieves the IDs of posts for the user. Then it reads each post using the ID retrieved from the cache.
* Each operation of adding a new post finds the friend feeds of every user subscribed to the post author, using the user ID, and renews the index for each of these posts. The index of the user’s blog is also renewed.
* Each operation of adding a new comment renews the index of the parent post.

### Test scenarios

There were two test scenarios:

1. Writing scenario:
   * Log in
   * View user’s blog
   * Add a new post
   * Log off
2. The more complex Viewing scenario:
   * Log in
   * View my blog
   * View friend feed, choose random post and view it, comment it with 10% probability; repeat five times
   * View my blog, choose random post and view it, comment it with 10% probability; repeat five times
   * Log off

### Performance tests

We tested for performance using the blog engine application with two data sizes (small and large). Every test run had a 10% probability of being a writing scenario and a 90% probability of being a viewing scenario. The test client had 32 threads that ran the two test scenarios. Such probabilities ensured writes to both Database and Velocity.

Tests conducted included:

* Throughput and Latency tests with and without Velocity
* Scalability tests with and without Velocity

For the Velocity version of the application, two additional tests were performed:

* Measure High Availability (HA) overhead
* Test Velocity failover functionality

### Throughput and latency tests

Throughput and Latency Tests were conducted using 6 Velocity cache nodes.

Small data sets

For the small data case, most objects in the Velocity cache were 4 KB. The database was about 22.5 GB with 16 GB of RAM on the Database. Velocity’s memory consumption in this case was reasonably constant, because all data was pre-cached. The Velocity processes consumed around 13 GB of memory as follows:

* Cache size = 11.65 GB
* Item count = 3.1M
* Region count = 1024

It is clear from this comparison that applications using Velocity enjoy significantly higher performance and lower latency than those that do not.

Figure 2: Throughput of applications with and without Velocity (4 KB objects)

Figure 3: Latency of applications with and without Velocity (4 KB objects)

Large data sets

For large data sets, most objects in the Velocity cache were 16 KB. The database was about 57 GB with 16 GB of RAM on the Database. Velocity’s memory consumption in this case was constant, but very high. Velocity processes consumed around 31 GB of memory, as follows:

* Cache size = 27 GB
* Item count = 3.1M
* Region count = 1024

For large data sets, the application with Velocity achieved throughput almost 15 times better than without it.

Figure 4: Throughput of applications with and without Velocity (16 KB objects)

Figure 5: Latency of applications with and without Velocity (16 KB objects)

### Scalability tests

To test scalability of the applications, up to 6 web servers were used in the web farm. To compare throughput and resource utilizations for each configuration, performance characteristics were measured when the latency reached a predefined threshold. These thresholds could be potentially different for applications with and without Velocity and for different object sizes.

Small data sets

As with the basic performance tests, small data tests used 4 KB objects to store in Velocity. Since the database size was only 22.5 GB and the Database RAM was 16 GB, almost all of the data could be loaded in Database memory. However, even with this object size, the application without Velocity did not achieve linear scalability. Introducing Velocity improved performance and scalability.

Figure 6: Scalability of applications with and without Velocity (4 KB objects)

Large data sets

Tests with 16 KB objects changed the performance characteristics of the applications significantly. The database became 57 GB and it could barely be stored in memory, which caused Database performance to degrade significantly. On the small web farms, Velocity could not store all the data in cache and this caused frequent calls to the Database, which degraded performance as well. However, very interesting things happened when increasing the web farm. The Velocity cache obtained more memory, more data was cached, and the Database load decreased. Such behavior allowed us to observe better than linear scalability of the application with Velocity.

Figure 7: Scalability of applications with and without Velocity (16 KB objects)

### Velocity vs. HA Velocity

High Availability tests were conducted using 6 Velocity cache nodes.

Small data sets

The performance of Velocity was not significantly different with HA turned off and on, although when turned on, Velocity consumes twice as much memory as without HA.

Figure 8: Throughput of applications with Velocity with HA turned on and off (4 KB objects)

Large data sets

In case of large data sets, Velocity with HA turned on cannot load all of the data into the cache. Many requests still require communication with the database server. As a result, the Velocity application with HA turned on had only half the throughput of the application with HA turned off.

Figure 9: Throughput of applications with Velocity with HA turned on and off (16 KB objects)

###### Failover tests

We performed failover tests with the blog engine application using 4 KB data and HA turned on. The following describes the test approach:

1. Run web farm with 6 web servers and generate a constant load
2. After some time turn off the first Velocity host
3. Wait while the application resumes its normal work
4. Turn off the second Velocity host
5. Wait while the application resumes its normal work
6. Turn on the first host
7. Wait until the application performance stabilizes
8. Turn on the second host
9. Wait until the application performance stabilizes

Cache hosts were stopped with the Velocity Admin Tool. The timeline for forced failures was the following:

1. At ~5:00 turn off the first host
2. At ~7:00 turn off the second host
3. At ~17:00 turn on the first host
4. At ~28:00 turn on the second host

Our tests indicated how failover works, displaying patterns for associated performance drops, and how Velocity recovers from node failures.

1st node added

2nd node added

2nd node removed

1st node removed

Figure 10: Throughput of application with failover testing (4 KB objects)

Major throughput reduction occurs when hosts are turned off. We also noticed that Velocity recovers from one host shutdown much better than from the second one. This is because there is less memory and fewer resources to overcome this shutdown. Response time shows the same trends. Less significant throughput reduction also occurs when adding nodes back to the cluster (repartitioning of data occurs when a new node is added). It is important to notice that these tests were performed with one DataCacheFactory per thread (thread-static). Experiments revealed some issues with the Velocity client, which caused one DataCacheFactory per process to perform worse during failover than one DataCacheFactory per thread. These issues affect only failover performance and Microsoft is expected to work on them in future versions.

Figure 11: Latency of application with failover testing (4 KB objects)

Network consumption shows how Velocity recognizes node failures and performs repartitioning.

Figure 12: Average web farm network usage of application with failover testing (4 KB objects)

CPU and memory consumption clearly explains how Velocity manages failover and repartitioning.

Figure 13: Web farm CPU usage of application with failover testing (4 KB objects)

Figure 14: Web farm memory consumption of application with failover testing (4 KB objects)

The first two repartitions can be seen at 5 and 7 minutes from the start. Memory consumption of the shut down nodes falls to zero, secondary regions become primary on rest of the nodes, and new secondary regions are created across the cluster. When nodes are added to the cluster, the memory consumption of these nodes rises smoothly. Memory consumption on other nodes most likely rises because some new partitions are added to these nodes, but old data resides for some time as garbage. This explains reducing memory consumption some time after adding the first host, which was caused by garbage collect cycle on some nodes.

Additional charts that show CPU and Disk usage are in Appendix A.

### Lessons learned

The benchmark tests of the blog engine application resulted in a number of interesting observations:

* Velocity provides linear scalability.
* It is important to estimate the throughput requirements for the application and to size the cache cluster accordingly.
* Adding Velocity can significantly reduce the CPU usage for the database layer.
* In addition to reducing CPU usage on the database, Velocity reduces CPU usage on the client, because it eliminates the need to prepare, run, and parse results from SQL queries.
* Expiration time on Velocity should not be set to very small values, which lead to increasing memory consumption, not decreasing. This occurs because of delays of garbage collection calls―old data expires, but remains in memory, and a new copy of data is added to memory.
* Velocity may scale better than linearly when scaling up from a small number of memory-poor nodes to store most-used data. Increasing the number of Velocity nodes increases the amount of memory available for most-used data and eliminates the database bottleneck.
* Velocity performs better in homogeneous web farm environments that distribute RAM equally among cache hosts.
* Velocity failover works, but a significant drop in application performance should be expected during failover.
* Adding new nodes to an existing cluster works smoothly and is less intrusive when compared to failed nodes.
* If failover performance is critical, consider using one DataCacheFactory per thread (thread-static) instead of using one global process-wide DataCacheFactory.

## Velocity Session State Provider

An e-commerce application is a good use case for demonstrating session state features. This application was used to compare performance between Database and Velocity session state providers. It simulated an online shopping site with simple functionality focused on adding products to a shopping basket, for which a session state provider was used. The final purchase was simulated by clearing the session state for the user.

### Architecture

The architecture of the e-commerce application can be summarized as follows:

* A Database with application data, containing product categories and products.
* Horizontally scalable web servers with basic functions of searching for an item, viewing the item, adding an item in the category, and deciding on the final purchase.
* (non-Velocity case) Each request to the web server generates a read and/or write request to session data, which is provided by the Database Session Provider.
* (Velocity case) Each request to the web server generates a read and/or write request to session data, which is provided by the Velocity Session Provider.



Figure 15: E-Commerce Application Architecture

The database session state provider used for the tests was a SQL Server Session state provider. Session state database was configured to use a custom database.

The specific hardware and database configurations used in the tests are shown in Appendix B.

### Test scenarios

Tests were conducted for a buying scenario, which consisted of the following steps:

* Go to the categories page, which displays two random categories.
* For each category, view products for that category page. The products page displays 5 random products.
* For each product, visit the product buying page, where the purchased product is added to the session state.
* After buying the product, visit the view cart page, where the session state is retrieved and products selected so far are displayed.
* After all products in all categories are selected, proceed to the check out page, which performs session cleanup.

Every test added 10 products to a session and viewed the session 10 times. Each test covered object sizes in Velocity from 1x product size to 10x product size.

###### Performance tests

The e-commerce application was run using a Database Session State Provider and Velocity Session State Provider on six web servers. We conducted our tests with four different ranges of object sizes. Tests conducted include:

* Throughput tests with Database and Velocity versions of the application
* Scalability tests with Database and Velocity versions of the application

For Velocity version of the applications two additional tests were performed:

* Measure High Availability (HA) overhead
* Test Velocity failover

###### Throughput tests

Our throughput tests indicated that, depending on data size, the application with Velocity either achieves parity with the Database or performs much better. The larger the objects, the better the Velocity session state provider performs in comparison to the Database session state provider.

Tests results indicate that smallest and small sized data show similar behavior and charts for the smallest are not included.

Tests were conducted using 6 nodes.

Figure 16: Throughput of application with Velocity and Database session state providers  
(3 KB object size)

Figure 17: Throughput of application with Velocity and Database session state providers  
(9 KB object size)

Figure 18: Throughput of application with Velocity and Database session state providers  
(30 KB object size)

###### Scalability comparison

To test scalability of the applications, up to 6 web servers were used in the web farm. The performance test was started for each configuration and waited for a predefined latency threshold. Latency started near 0 and began increasing only after the throughput reached a plateau. At this moment, performance characteristics were averaged and saved. The latency threshold may differ for different object sizes and session state providers, but for session state it was chosen to be around 1.5 seconds.

Test results indicate that smallest and small sized data show similar behavior and charts for the smallest are not included.

Figure 19: Scalability of applications with Velocity and Database session state providers  
(3 KB object size)

Figure 20: Scalability of applications with Velocity and Database session state providers  
(9 KB object size)

Figure 21: Scalability of applications with Velocity and Database session state providers  
(30 KB object size)

Only the final two comparisons showed any significant deviation, as the Database scaled well using small objects. As the object sizes increased, the Database performance did not flatten out―it decreased. One of the possible explanations for this observed behavior is that the Database became overloaded with requests from web servers. New requests come before old requests are serviced and this degrades performance. The bottleneck in the Database becomes apparent beginning with 3 web servers. The influence of this database disk bottleneck on the web farm is clear―web servers were no longer fully utilized (Appendix B has the disk usage charts).

###### Velocity vs. HA Velocity

As was the case for the blog engine application, there is no major difference between Velocity with HA turned on and off when all the data can fit in the cache memory. In general, the heavier objects are, the more significant the difference is.

Tests results indicate that smallest and small sized data show similar behavior and charts for the smallest are not included.

Tests were conducted using 6 Velocity cache nodes.

Figure 22: Throughput and latency of Velocity with HA turned on and off (3 KB object size)

Figure 23: Throughput and latency of Velocity with HA turned on and off (9 KB object size)

Figure 24: Throughput and latency of Velocity with HA turned on and off (30 KB object size)

###### Failover

We performed failover tests with the Velocity session state application using the large object size and HA turned on. The following describes the test approach:

* Run a web farm with 6 web servers
* Turn off one Velocity host
* Wait until the application resumes its work
* Turn off a second Velocity host
* Wait until the application resumes its work

Cache failure was simulated using three approaches: Process Kill, Service Stop, and using the Velocity Admin Tool to stop the Cache hosts. All three produced remarkably similar results; only the graph for Process Kill is shown to illustrate the effect of forced failures on throughput and latency. The timeline for forced failures was as follows:

1. At ~4:30 turn off the first host
2. At ~9:30 turn off the second host

Figure 25: Application with Velocity failover by process killing (30 KB object size)

### Lessons learned

The session state benchmarking resulted in a number of interesting observations:

* Velocity provides linear scalability.
* Velocity can run on inexpensive machines and provide session state functionality that is scalable and highly available.
* The Database may demonstrate negative scalability because of flooding with requests from additional web servers.
* Velocity may remove bottlenecks from the Database in different ways: CPU usage, disk usage, and even network usage.
* Even if the Velocity session state provider works only slightly better than the Database provider, it significantly reduces the load on the database servers. This is particularly useful if session state information and application domain data are stored in the same database.

## Velocity Notifications

The notifications feature of Velocity is well suited for implementing event-driven applications, such as one that processes incoming stock market data. In this application profile Velocity works as a repository of market messages that are fed by a loader and these are consumed by trade consumer software. Each consumer works with one or more instruments and is subscribed to the changes to the instruments. For each trade, the consumer client calculates an overall trading volume for that instrument.

This application profile, unlike the previous two use cases, was implemented with Velocity only, as the goal was to measure the performance characteristics of notifications and notification clients.

### Architecture

The architecture of this application can be summarized as follows:

* Load Agents generate trades and write them into a Velocity cache cluster.
* A number of event consumers (referred to as listeners in the diagram), configured as Windows Services are subscribed to specific event types.
* On receiving an event notification, the consumer takes an action dictated by the business logic.



Figure 26: Market Data Application Architecture

### Velocity usage

The Velocity cache was used with custom regions, each of which contained trades for one instrument. The load generator simulated trades and inserted items into the cache. Event consumers were installed on the same hosts as Velocity. Each consumer program listened for events from a subset of regions. Each region was handled by only one consumer. Regions were distributed among consumers during application startup.

By default, Velocity sends notifications in bulk in a single thread. In order to not backup these threads and block future notifications, it was important to free up this thread as soon as possible. To accomplish this, we introduced a thread pool that gives the ability to parallelize processing and asynchronous processing. The detailed algorithm is as follows:

* Client enumerates through all notifications received from Velocity
* For each notification, the program queues the user work item in the thread pool
* Each item in the thread pool does the following:
  + Retrieves the trading data item from Velocity
  + Calculates internal statistics to be logged in another thread
  + Calls custom processing pieces that were subscribed to this instrument

In theory, there could be many custom processing logic pieces, each of which could compute private metrics and perform a real-time portfolio analysis. For this application, there was only one system piece, which calculated overall trading volume for a particular instrument. For each data item, the trading volume calculator added the volume of this item to a locally cached sum for the instrument. A separate thread updated volumes of all instruments handled by the client to a Velocity cache several times a second. This change was done to improve client performance. Overall trading volumes for each instrument were cached in the default region of the Velocity cache.

### Test scenarios

For the notifications application, only one test scenario was used. This scenario, run by the load agents, represented a market data source and added random trades to the cache. For the benchmark tests 100 regions/instruments were used.

### Performance tests

The Notification application was run on six servers. Each of these servers was running a Velocity cache. Both throughput and scalability tests were conducted.

###### Throughput test

For the Notifications application, latency is not a key metric. When consumers are able to handle all notifications, latency is almost equal to or less than the notifications send interval (as set in the Velocity configuration). The important metric for this application is throughput. Maximum throughput is important as this decides when the client will start falling behind the sends; once this happens latencies can increase dramatically.

We measured cache item generation and the notification receiving rate to determine maximum throughput. When sent and received event rates begin to diverge, that point indicates the maximum throughput rate.

Figure 27: Throughput of application on 6 nodes

Figure 27 shows that the maximum throughput of the application on 6 Velocity nodes is about 24,000. After this point, notifications begin to lag and Velocity begins to use more processor time. The client application, in contrast, begins to handle fewer notifications because of this. As soon as generation is stopped, the client can catch up on its unhandled notifications and the client receiving speed increases, because Velocity frees up some CPU, which then becomes available to the client. The CTP4 version of Velocity will introduce bulk notifications from regions/cache, which could further improve performance of notifications.

Figure 28: CPU usage across the server farm of application on 6 nodes

This observation can be seen more clearly when running application on one node.

Figure 29: Throughput of application on 1 node

The CPU usage charts also show the trends mentioned above.

Figure 30: CPU usage across the server farm of application on 1 node

There were also some tests performed with the different data size, when the Data field was set to array of 512B. With this size, Velocity performs the same as with an empty Data field.

Figure 31: Throughput comparison of application on 6 nodes

Some tests with bigger data were also tried and the performance was almost the same until all cache items fit in memory.

###### Scalability test

To test scalability of the notifications, we used cache clusters up to 6 nodes. For each cluster size the maximum was derived using the pattern mentioned above (when notification receives start to fall behind the sends). Our results indicate that the notifications scaled linearly with all data sizes.

Figure 32: Scalability of market data application

### Lessons learned

The notification benchmark tests resulted in a number of interesting observations:

* Velocity provides linear scalability for the notification processing application.
* For applications using Velocity notifications, it is important to find the maximum throughput and to monitor performance so as to not generate more events than the observed maximum limit. Generating more notifications decreases client performance and might even cause notification loss (if the backlog is too big that cache cannot hold everything).
* If the Velocity cluster has enough memory to handle all data, increasing object sizes do not decrease performance significantly.

# Creating a Real-World Application

As part of the benchmarking tests we observed how each of the sample applications improved its performance by using a specific Velocity feature. A typical enterprise application is much more complex and has multi-dimensional performance issues when compared to the benchmark applications. Such an application is usually faced with a web layer that needs to be scaled according to traffic, searches that rely on complex queries from a database, frequent reads and writes to a database, and synchronous execution of long running tasks. The versatility of Velocity and its rich feature set makes it a good fit for solving all of these problems easily. A full e-commerce application is a typical application confronting all of these challenges.

Every online shop uses shopping carts and these are stored in the session state of the web application. Managing session state is not a problem when the web front end consists of only one server. In scalable web applications there is a web farm that grows or shrinks based on traffic patterns. In these cases sessions cannot be associated with the web server but need to be stored in a separate location so that they are still accessible when the web servers are brought up or down. This location is frequently provided by a database, such as SQL Server. From the Session State Provider performance results, it is clear that the Velocity session state provider scales and performs much better than the Database provider. Velocity’s high availability features ensure that the session information is available even in case of a cache server failure. This feature is not easy to implement with a database.

A typical online shop will experience many searches for products using different criteria, which will involve complex queries to the database. Running many complex queries will overload the database server and degrade the performance of the application. Another commonly observed pattern is that there are products that are popular on a specific day because of sales promotions. In such cases, these items are searched by many people and the same items are queried from the database again and again. As was shown in the Distributed Cache tests, Velocity can be used to cache most-used data, thereby reducing database overhead. In this case search indexes, promotional items, and popular items can be cached. Caching these items will effectively free the database from handling lots of similar requests. Since rebuilding indexes is an expensive process (with significant database load) it is better to keep them always cached. Velocity’s high availability feature ensures that these items will be available even in case of a failure of one of the cache servers.

The performance of most applications can be improved by making long running tasks asynchronous and parallel. Lots of technologies exist to allow developers to implement event-driven applications. Velocity brings event-driven distributed programming alongside distributed storage of data, which simplifies parallelization of asynchronous processes. The online shop application is not an exception and it also has a number of operations that can be made asynchronous. The most important operations are related to rebuilding product search indexes based on some events, such as adding a new promotional product, when a product rating is changed by user, and when a product is purchased by a user. The Notifications tests show that Velocity has an effective notifications mechanism. Notifications are the best way to asynchronously trigger a long running task, such as rebuilding search indexes based on popularity, ratings, etc., or creating a list of frequently accessed items, which not only eliminate bottlenecks but also improve the user experience.

A common architecture for such a scalable application is shown in Figure 27.



Figure 33: Real-World Scalable Application Architecture

# Feature Comparisons

Grid Dynamics compared Velocity features with three other well known distributed caches and web scalability products. The comparison was performed using Velocity (CTP3 Release), Oracle's Coherence Grid Edition (3.4.2), the open source *memcached* (1.2.8), and Terracotta FX Edition (3.0). Some of the features mentioned here may not be available in other editions of these products. Although a line by line comparison of Velocity features to the companion products or technologies reveals that the current version of Velocity does not address every conceivable feature, developers should evaluate the requirements of their application and their preferred programming environment against the relative costs of each of these technologies.

| **Feature** | **Velocity** | **Coherence** | **memcached** | **Terracotta** |
| --- | --- | --- | --- | --- |
| **API** |  |  |  |  |
| **CRUD Operations (Create, Read, Update and Delete)** |  |  |  |  |
| **Cache Object type** | Serializable CLR Object, Data Contracts, Byte[] | Multiple serialization formats: Serializable, Externalizable, ExternalizableLite, POF (POF is high-density proprietary format) | API converts objects to String | Any Java Object |
| **Supported Client Languages** | .NET Languages | Java, C++, C#, VB.NET, Delphi, Jscript | Perl, C, C#, PHP, Python, Java, Ruby, and PostgreSQL | Java |
| **Named Cache support** |  |  |  |  |
| **Tag Based query[[1]](#footnote-1)** |  |  |  |  |
| **Concurrency API** | Optimistic & Pessimistic Locking | Optimistic & Pessimistic Locking. Transactions support |  | Supported using Java Concurrency Semantics |
| **Continuous Query support** |  |  |  |  |
| **Cache Notifications** | [[2]](#footnote-2) |  |  |  |
| **Eviction** |  | [[3]](#footnote-3) |  |  |
| **Persistence** |  |  |  |  |
| **IDE Integration** |  |  |  |  |
| **Supported IDE** | Visual Studio | Visual Studio, Eclipse |  | Eclipse |
| **Supported Topologies** |  |  |  |  |
| **Partitioned** |  |  |  | [[4]](#footnote-4) |
| **Primary/Backup** |  |  |  |  |
| **Replicated** |  |  |  |  |
| **Near Cache/Local Cache** |  |  |  |  |
| **Dynamic Cluster Membership** |  |  |  |  |
| **Sessions** |  |  |  |  |
| **.NET Session State Provider** |  |  | [[5]](#footnote-5) |  |
| **Java Sessions** |  |  |  |  |
| **Failover** |  |  |  |  |
| **Failover Support (High Availability)** |  |  |  |  |
| **Security** |  |  |  |  |
| **Secure Cache Support** |  |  |  |  |
| **Monitoring & Administration** |  |  |  |  |
| **Command Line Tooling** |  |  |  |  |
| **Cluster wide management** | [[6]](#footnote-6) |  |  |  |
| **GUI to monitor cluster and clients** |  | Third party tooling[[7]](#footnote-7) |  |  |
| **Integration with external monitoring tools** | ETW, PerfMon | JMX based integration |  | JMX based integration |
| **Extensibility[[8]](#footnote-8)** |  |  |  |  |
| **Read Through, Refresh Ahead, Write Behind, and Write Through** |  |  |  |  |
| **Explicit Data Affinity** | [[9]](#footnote-9) |  |  |  |
| **Cache Embedded code** |  | [[10]](#footnote-10) |  |  |
| **Other** |  |  |  |  |
| **Object Size Limit** | None | None | 1 megabytes | None |
| **Cache Composition** | Client can be any supported language client. Server node is .NET runtime | Client can be any supported language client. Server node is a Java process | Client can be any supported language. Server node is a C++ process | Client and Server are Java processes |
| **Product Editions** | Part of your Windows investment | Standard, Enterprise and Grid (Commerial License) | Single edition with all features (free) | ES (Free), EX and FX (Commerical License) |

# Conclusions and Recommendations

The Microsoft Velocity distributed in memory application cache platform is an effective means of improving the performance and scalability of enterprise applications. Benchmark results demonstrate that in real world scenarios, it significantly outperforms applications built only with database access. It provides linear scalability under loads that cause non-Velocity applications to bottleneck and suffer performance degradation.

In the blog engine and e-commerce website applications, Velocity provided linear scalability, improving performance as the number of web servers in the web farm was increased. Database-only applications may demonstrate negative scalability because of flooding with requests from additional web servers. Velocity can remove bottlenecks from the Database in various ways: CPU usage, disk usage, and even network usage.

Although the Velocity session state provider in the e-commerce website test worked only slightly better than the Database provider for smaller data, it offloaded the database servers significantly. For large sized objects Velocity clearly shows significant performance gains.

For applications using Velocity notifications, it is important to find maximum throughput and to not to generate more events than observed maximum limits. Generating more notifications will decrease client performance and might even cause loss of notifications.

# Appendix A: Blog Engine Additional Details

### Hardware Setup

The Blog Engine application used the following configuration:

|  |  |  |
| --- | --- | --- |
| **Name** | **Hardware** | **Software** |
| **Web1** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web2** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web3** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web4** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web5** | 4 cores, 8 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web6** | 4 cores, 8 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Sql1** | 8 cores, 32 RAM (limited to 4x16) | SQL Server 2008 |
| **Load1** | 8 cores, 32 RAM | DNS, AD, 4 VMs with Load Agents |
| **Load2** | 2 cores, 8 RAM | Load Agent |
| **Load3** | 2 cores, 8 RAM | Load Agent |

Preliminary results were gathered on a different configuration with a more heterogeneous web farm. Tests have shown that the web servers’ bottleneck is CPU and scalability is better demonstrated on the servers with the same number of cores. This is the main reason that our final configuration used web servers with the same number of cores.

The Database was limited to 4 cores and 16 GB RAM to better fit our test lab and obtain real-world database performance characteristics. With 32 GB, the Database with small test data for the blog engine completely fit in memory, so the performance test results were not accurate. The same is true for CPU. With 8 cores, 6 web servers were not able to overload the Database. Generally, the only difference between limited and not limited is the number of web servers to send requests to the Database. Scalability will not be linear as well. Moreover, for session state testing, database CPU and RAM make only a small difference, because the bottleneck is the disk performance.

The network on all servers consisted of four 1-gigabit network cards, aggregated with link aggregation. So, network throughput was 4 gigabit/sec.

### Database Details

For the blog engine, we tested two different object sizes: 4 KB and 16 KB. Talking about object size the blog engine sample is not generally correct, because of specifics of Velocity cache usage. However, most of objects in the cache were the appropriate size. For example, 1 blog post contains ~4 KB data in small database and a list of 20 comments also contains ~4 KB of data. See “Velocity usage pattern” for details.

The database structure and number of records were the same for both object sizes:

|  |
| --- |
| **Database characteristics** |
| **75K users** |
| **~40 friends per user** |
| **1.5M posts** |
| **30M comments** |

With these characteristics, the Database for small data used 22.5 GB on disk, and Database for big data used 57 GB on disk. Friends per user, posts per user, and comments per ports ratios were random, but the probability distributions were equal for all databases. The distribution was chosen to better simulate a real-world application and to obtain the best performance results on the given hardware configuration.

A concrete number of users, and, hence, size of databases, was chosen to show the most interesting aspects of SQL and Velocity performance on the given hardware configuration.

### Database usage pattern

Typical queries to the database included requesting a user’s posts, requesting a user’s friend feed, requesting a blog post with comments, adding a new post, and adding a new comment. We expected a major load on the database in the field of requesting a friend feed with the query:

var q = from f in context.Friends

where f.User.UserName == userName

from p in context.Posts

where p.UserId == f.FriendId

join c in context.Comments on p.Id equals c.PostId into g

orderby p.Date descending

select new PostData(p.Id, p.Date)

.SetData(p)

.SetUser(p.User)

.SetCommentCount(g.Count())

To perform heavier reads, which can be cached with Velocity, we also tried more complex friend feed queries to display friend feeds, assembled by latest friend comments, such as on friendfeed.com. Unfortunately, in this case, a query to the database became so complex that it required 20 seconds to execute on large database:

var q = from f in context.Friends

where f.User.UserName == userName

from c in context.Comments

where c.UserId == f.FriendId

group c by c.Post

into g

orderby g.Max(c2=>c2.Date) descending

select new PostData(g.Key.Id, g.Key.Date)

.SetData(g.Key)

.SetCommentCount(g.Count())

.SetUser(g.Key.User)

.SetLastCommentDate(g.Max(c2=>c2.Date));

### SQL Queries

###### Simple friend feed

SELECT TOP (20) [t5].[PostId] AS [postId], [t5].[Date] AS [postDate], [t5].[UserId], [t5].[Title], [t5].[Text], [t5].[UserId2] AS [Id], [t5].[UserName], [t5].[value] AS [count]

FROM (

SELECT [t2].[PostId], [t2].[Date], [t2].[UserId], [t2].[Title], [t2].[Text], [t3].[UserId] AS [UserId2], [t3].[UserName], (

SELECT COUNT(\*)

FROM [dbo].[Comments] AS [t4]

WHERE [t2].[PostId] = [t4].[PostId]

) AS [value], [t0].[FriendId], [t1].[UserName] AS [UserName2]

FROM [dbo].[Friends] AS [t0]

INNER JOIN [dbo].[aspnet\_Users] AS [t1] ON [t1].[UserId] = [t0].[UserId]

CROSS JOIN ([dbo].[Posts] AS [t2]

INNER JOIN [dbo].[aspnet\_Users] AS [t3] ON [t3].[UserId] = [t2].[UserId])

) AS [t5]

WHERE ([t5].[UserId] = [t5].[FriendId]) AND ([t5].[UserName2] = @p0)

ORDER BY [t5].[Date] DESC',N'@p0 nvarchar(5)',@p0=N'user1

###### Complex friend feed (by comments)

SELECT TOP (20) [t4].[PostId] AS [postId], [t4].[Date] AS [postDate], [t4].[UserId], [t4].[Title], [t4].[Text], [t3].[value2] AS [count], [t5].[UserId] AS [Id], [t5].[UserName], [t3].[value3] AS [date]

FROM (

SELECT MAX([t2].[Date]) AS [value], COUNT(\*) AS [value2], MAX([t2].[Date]) AS [value3], [t2].[PostId]

FROM [dbo].[Friends] AS [t0]

INNER JOIN [dbo].[aspnet\_Users] AS [t1] ON [t1].[UserId] = [t0].[UserId]

CROSS JOIN [dbo].[Comments] AS [t2]

WHERE ([t2].[UserId] = [t0].[FriendId]) AND ([t1].[UserName] = @p0)

GROUP BY [t2].[PostId]

) AS [t3]

INNER JOIN [dbo].[Posts] AS [t4] ON [t4].[PostId] = [t3].[PostId]

INNER JOIN [dbo].[aspnet\_Users] AS [t5] ON [t5].[UserId] = [t4].[UserId]

ORDER BY [t3].[value] DESC',N'@p0 nvarchar(5)',@p0=N'user1

### Additional Charts

###### Throughput and latency tests

Small data set

CPU of the Database is overloaded when the application doesn’t use Velocity. Even using Velocity, the database server heavily uses CPU for writing.

Figure A1: Database CPU usage of applications with and without Velocity (4 KB objects)

For the application without Velocity, it is clear that the bottleneck is in the Database CPU―average web farm CPU usage is far from its limit. In the case of the Velocity enabled application, CPU of the web farm is near its maximum.

Figure A2: Web farm CPU usage of applications with and without Velocity (4 KB objects)

Memory usage was pretty much constant during the test. As mentioned previously, it is very small on web servers, because the size of the Velocity cache is not very big. In contrast, the Database consumes all available memory (16 GB), as always.

Large data set

Figure A3: CPU usage on web farm and Database of application with HA turned on  
(16 KB objects)

Figure A4: Disk usage on Database of application with HA turned on (16 KB objects)

Disk percentage usage on this and the following graphics was calculated by a Disk Idle time performance counter.

###### Scalability Tests

Small data set

It is clear that the bottleneck was in the database. For the application without Velocity, Database reached 100% usage of disk and CPU. In contrast, applications with Velocity did not load the database a lot and put most of the load on web servers to allow them to scale linearly.

Figure 28: Database CPU usage comparison of applications with and without Velocity  
(4 KB objects)

Figure A6: Web farm CPU usage comparison of applications with and without Velocity  
(4 KB objects)

Figure A7: Database disk usage comparison of applications with and without Velocity  
(4 KB objects)

Large data set

The process of freeing Database resources in the application with Velocity can be shown in the following diagrams:

Figure A8: Database CPU usage for applications with and without Velocity (16 KB objects)

Figure A9: Database disk usage for applications with and without Velocity (16 KB objects)

Even the web server fully loads the Database state. In contrast, web servers became more loaded while the Database bottleneck was being removed.

Figure A10: Web farm CPU usage for applications with and without Velocity (16 KB objects)

In the tests with big data, network usage also increased. However, the major network load was observed not on the web servers, but on the load balancer. As in the case with small data, Velocity network usage was still minimal.

# Appendix B: E-Commerce Website Details

### Hardware Setup

The e-commerce website used the following configuration:

|  |  |  |
| --- | --- | --- |
| **Name** | **Hardware** | **Software** |
| **Web1** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web2** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web3** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web4** | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web5** | 4 cores, 8 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Web6** | 4 cores, 8 RAM | IIS, .NET 3.5 SP1, ASP.NET, Velocity |
| **Sql1** | 8 cores, 32 RAM (limited to 4x16) | SQL Server 2008 |
| **Load1** | 8 cores, 32 RAM | DNS, AD, 4 VMs with Load Agents |
| **Load2** | 2 cores, 8 RAM | Load Agent |
| **Load3** | 2 cores, 8 RAM | Load Agent |

The network on all servers consisted of a four 1-gigabit network cards, aggregated with link aggregation, resulting in 4 gigabit throughput.

### Database Details

For session state testing, we used a database with products and product categories. However, for this application, the e-commerce part of the database is not important and exists just to simulate a typical e-commerce scenario. The database is relatively small in order to get clearer session state performance numbers. Product database statistics are the following:

* 712 MB on disk
* 40 product categories
* 4020 products

These numbers include products of several different sizes, to test session state performance in different scenarios: 300 bytes, 3 KB, 9 KB, and 30 KB. Different sizes of products were used in different tests, but were located in a single database to simplify the process of generating test data.

### Database usage pattern

Database usage was oversimplified to provide clear performance results for the session state provider. All queries were either queries of entities by ID or queries for products by category ID.

### Session state usage pattern

The application emulated users of an e-commerce website, which can add products to a shopping cart and then either buy products or refuse to buy and clean the cart. For the sake of simplicity and measurement clarity the actual buying process was not emulated. After the user finishes shopping, the application just clears the shopping cart.

To emulate different object sizes in the session state, the application puts the entire product object in the session.

Unlike the preceding blog engine application, logging is not required for implementing this e-commerce website.

### Additional Charts

###### Throughput Tests

Peaks of the application with the Database session state provider are explained by disk performance peaks. In these moments, heavy disk write operations occur. Most likely, these operations are related to copying data from the transaction log to the main database file. The application with Velocity session state always performed better, but the real improvement can be seen starting from 9-90 KB session state sizes. On these sizes, the Database disk became overloaded, while Velocity still works well. On the largest data, Database performance becomes unpredictable due to disk writing peaks. These peaks can be easily noticed from disk usage graphics.

Figure B1: SQK Server disk performance of application with Velocity and Database session state providers (3 KB object size)

Figure B2: SQK Server disk performance of application with Velocity and Database session state providers (9 KB object size)

Figure B3: SQK Server disk performance of application with Velocity and Database session state providers (30 KB object size)

In the case of session state providers, disk is the main bottleneck on Database. CPU on the database server is not used significantly. On the web farm, in contrast, CPU is still the most utilized resource. This fact may help a lot in investigation of the bottlenecks. If web CPU is at its maximum, Database performs well. If web farm CPU decreases, Database is overloaded and it becomes a bottleneck.

Figure B4: CPU usage of application with Velocity and Database session state providers  
(3 KB object size)

Figure B5: CPU usage of application with Velocity and Database session state providers  
(9 KB object size)

Figure B6: CPU usage of application with Velocity and Database session state providers  
(30 KB object size)

Network usage of the application with Velocity session state is high for large data, but less than a 1 gigabit/sec. In contrast, the application with Database session state uses network on the database very heavily. In the peak load, it is about 1.5 gigabit/sec.

Figure B7: Network usage of application with Velocity session state provider  
(30 KB object size)

Figure B8: Network usage of application with Database session state provider  
(30 KB object size)

### Scalability Tests

As shown previously, the bottleneck is in Database and it shows itself beginning with 3 web servers. The influence of this bottleneck on web CPU is clear: the web servers became underloaded.

Figure B9: CPU usage of applications with Velocity and Database session state providers  
(9 KB object size)

Figure B10: CPU usage of applications with Velocity and Database session state providers  
(30 KB object size)

Figure B11: Database disk usage of applications with Velocity and Database session state providers (9 KB object size)

Figure B12: Database disk usage of applications with Velocity and Database session state providers (30 KB object size)

# Appendix C: Market Data Application Details

### Hardware Setup

The hardware lab configuration for the Market Data Application, which measured Velocity notifications throughput, is a bit different from the lab configuration for the other test applications.

|  |  |  |
| --- | --- | --- |
| Name | Hardware | Software |
| App1 | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, Velocity |
| App2 | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, Velocity |
| App3 | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, Velocity |
| App4 | 4 cores, 16 RAM | IIS, .NET 3.5 SP1, Velocity |
| App5 | 4 cores, 8 RAM | IIS, .NET 3.5 SP1, Velocity |
| App6 | 4 cores, 8 RAM | IIS, .NET 3.5 SP1, Velocity |
| Load1 | 8 cores, 32 RAM | 4 VMs with Load Agents |
| Load2 | 8 cores, 32 RAM | DNS, AD, 4 VMs with Load Agents |
| Load3 | 2 cores, 8 RAM | Load Agent |
| Load4 | 2 cores, 8 RAM | Load Agent |

This application required more load agents, but Database was no longer required. The web farm remained the same, but for this application it is better to call it an application server farm, because there was no web interface.

The network was 1 gigabit/sec on all machines.

### Cache Data Structure

Cache data item representing a trade used the following structure:

    public class Data  
    {  
        public DateTime Date;  
        public int Count;  
        public decimal Cost;  
        public byte[] Data;  
    }

The serialized class was very tiny, so to experiment with different data sizes, an optional data parameter was used.

# About Grid Dynamics

Grid Dynamics is the global leader in scaling mission-critical systems. We help customers architect, design, and deliver business systems that handle peak loads, scale on demand, and always stay up. Using the latest advances in grid and cloud computing, our customers turn monolithic applications into scalable services and static, underutilized server clusters into virtualized compute clouds. The results: better performance, higher availability, faster time-to-market, and lower operational costs.

Grid Dynamics’ .NET Scalability Practice offers solutions that enable customers to develop and scale applications with the Velocity caching platform, Windows HPC Server 2008, the Azure Services Platform, and other Microsoft .NET technologies.

Grid Dynamics Consulting Services, Inc.

39300 Civic Center Drive, Suite 145

Fremont, CA 94538

Tel: +1 510-574-0872

Fax: +1 636-773-4528

<http://www.griddynamics.com/>

1. Tag based queries let you search for objects with a tag. Coherence Filter support can be used for this [↑](#footnote-ref-1)
2. Velocity notifications are polling based and not real time. Coherence supports both Synchronous and Asynchronous notifications [↑](#footnote-ref-2)
3. Coherence supports eviction by memory usage and number of objects in Cache [↑](#footnote-ref-3)
4. Terracotta does not have a partitioned cluster feature. Mirrors can stripe the data across multiple servers [↑](#footnote-ref-4)
5. A memcached session state provider is available as a codeplex project [↑](#footnote-ref-5)
6. Velocity administration is done using Windows PowerShell and includes a rich set of commands that support cluster-wide operations [↑](#footnote-ref-6)
7. Evident Software’s [ClearStone Live](http://www.evidentsoftware.com/products/clearstone_live.aspx) is a third-party tool that provides GUI monitoring [↑](#footnote-ref-7)
8. Extensibility in Terracotta is achieved using pre-packaged plug-ins (Terracotta Integration Modules) available as free downloads from Terracotta Forge [↑](#footnote-ref-8)
9. Named regions in Velocity can be used for Explicit Data Affinity [↑](#footnote-ref-9)
10. Coherence supports InvocableCache in .NET, EntryProcessor, and EntryAggregator. These allow node-based processing [↑](#footnote-ref-10)