Achieving the Impossible: Unlimited Application Scalability

An Oracle White Paper
Updated May 2007

ORACLE®

The most common challenge organizations face when it comes to achieving scalability isn't the cost of scalability itself but the difficulty of predicting that cost once applications grow beyond a certain threshold.

INTRODUCTION

For applications that need to scale to stratospheric levels, the prospect of maintaining 100 percent application availability while simultaneously growing a system to handle an ever-increasing load is very daunting. It becomes almost impossible when you add the business requirement of keeping the cost of incremental capacity constant.

The most common challenge organizations face when it comes to achieving scalability isn't the cost of scalability itself but the difficulty of predicting that cost once applications grow beyond a certain threshold. Organizations require not only application scalability but also the *predictability* of application scalability, in terms of both cost and effort.

For modern applications, particularly those built for the Java platform, there are proven approaches and solutions that can help an organization achieve predictable, scalable performance for their applications. And although these approaches may be most valuable for extreme-scale applications, they can save time and money in almost any server-based application.

This paper focuses on one specific approach—the use of clustered caching to provide applications with significantly higher throughput and lower latency for data operations while retaining the appropriate levels of data quality that the applications require.

DEFINING SCALABLE PERFORMANCE

It is important to start by understanding the problem. The following are some real-world examples:

- An application service provider (ASP) deployed an instance of its Java 2, Enterprise Edition (J2EE) application in a cluster for a large customer. Switching from the single-server environment to a dual-server environment made the overall application throughput drop significantly.
- At peak load, an internal Web-based human resources system would grind to a halt, because its dedicated 32-CPU database became saturated. Even worse, the application was serving mostly static content.

- A successful application service provider (ASP) had to stop signing up new customers, because, as the company grew, the incremental cost of adding customers increased until it was losing money every time it added a new one. Its market success would have meant bankruptcy.
- Although it provided near-instant response times in development and testing, a data-intensive mutual fund analysis application was averaging more than 15 seconds per page in production, due primarily to the database load caused by the number of concurrent application users.

In each of these cases, during application development, the development team had deemed that the applications worked correctly. Response times appeared to be well within acceptable bounds. It wasn't until production load was applied that the problems were realized. This is the difference between performance and scalable performance.

Performance refers to an application's ability to achieve a response within a certain period of time. This is called a wall clock measurement. Performance, which is quite simply the inverse of latency (the time delay between when data is requested and when it is received), is important because it reflects the responsiveness an end user will experience when using the application.

Scalable performance means that response times for an application are within defined tolerances for normal use and that they remain within those tolerances up to the expected peak user load.

Scalable performance means that response times for an application are within defined tolerances for normal use and that they remain within those tolerances up to the expected peak user load. It also requires a clear understanding of the resources that would be necessary to support additional load without exceeding those tolerances. Scalable performance is not just about performance; it is also about maintaining acceptable performance under load. Scalable performance is not focused on making an application faster. Rather, it ensures that application performance does not degrade below defined boundaries as the application load increases. It defines how resources must grow to ensure acceptable performance, and it provides the basis for predicting with certainty which specific additional resources will be required to handle the load.

If performance is about latency, such as how long it takes to respond to a hypertext transfer protocol (HTTP) request, then scalable performance is about throughput, such as how many HTTP requests an application can process per second.

PREDICTING SCALABLE PERFORMANCE

Organizations don't just require application scalability; they also require the *predictability* of application scalability. To predict how an application will behave under increasing load, it is important to understand the factors that affect scalable performance as load is increased.

The ultimate goal in predictability is to be able to understand the resource cost represented by the average user and to drive all of the other factors as close to zero as possible. In other words, if you quantify the resource utilization of each user, you

might be able to predict the amount of resources that would be necessary to support any given number of users.

The side effect of concurrency is most obvious when multiple concurrent operations work with shared resources.

The first side effect that makes such *prediction by multiplication* impossible is concurrency—the effect user actions have on each other. The side effect of concurrency is most obvious when multiple concurrent operations work with shared resources. There are many types of shared resources, including database constructs such as summary tables (tables that are automatically updated and that, by their nature, tend to have higher levels of contention) and application constructs such as synchronized methods on singleton objects.

Concurrency issues can degrade the ability to achieve scalable performance. In many applications, the concurrency issues result from a well-intentioned attempt to provide an absolutely correct answer in a situation that does not benefit from one. For example, inventory systems in online shopping applications do not benefit from being real-time and exact for two reasons: First, the quantity of items in inventory that would benefit from being absolutely correct is going to be the same quantity that will change faster than the user can respond. Second, and more important, any effort to maintain a real-time quantity for display purposes will be wasted, in view of the amount of time it takes the HTTP response to be communicated and presented to the end user.

The second side effect that makes scalable performance difficult to predict is the desire to make the various operations within an application flow through a similar series of steps. Quite often this means that every HTTP request is going to be load-balanced across a stateless farm of Web servers and routed to its sticky-designated J2EE application server for processing. The designated application server sends every request through the same series of processing steps, including a front controller, a request handler, a view generator, and so on. Each of these, in turn, may request data from the application's domain model, which itself will likely turn to the database for answers.

Although this type of generic application architecture may be easy to assemble by use of IDE wizards and a modern Web framework, applications that take this approach exhibit a high degree of intratier communication complexity. This means that each hop from one server to the next and back again adds to a complexity factor that directly impairs scalable performance.

More important, the cost of each tier in the application tends to increase dramatically. Consider this:

- In the first tier—the Web tier—the cost of serving static content by use of commodity boxes running Apache is very low and, due to its stateless nature, very predictable.
- In the second tier—the J2EE tier—the cost of hardware may be the same, but both the infrastructure software and the custom application logic are significantly more expensive.

- In the third tier—the database tier—the cost of the hardware and software is often significantly higher and the cost of scalability tends to grow exponentially.

The result is predictable: Applications that tend to allow each request to get past the first or second tier will be much more expensive to scale, and applications that rely on data access from the database for each request will encounter severe limits on their scalable performance.

The lesson learned is this: handle the incoming load in the earliest-possible tier, and do it with resources that have predictable scale-out behavior and costs. For most applications, those are the resources represented by commodity hardware—servers with fast CPUs and plenty of memory.

Also, you should deliberately build an architecture that will bottleneck in the earliest-possible tier on either the CPU or in memory. Why? You're going to have a bottleneck, whether you choose to or not. And if you don't pick the bottleneck deliberately, it's going to end up in the most expensive and difficult-to-scale part of your environment.

Clustering for Scalability

The term *scaling out* is used to explicitly contrast with the traditional approach to scaling, called *scaling up*—more commonly known as buying bigger boxes.

There are three distinct tiers in the previous example, and each has the ability to scale out. The term *scaling out* is used to explicitly contrast with the traditional approach to scaling, called *scaling up*—more commonly known as buying bigger boxes. Scaling out achieves scalability through the purchase of more boxes of the same type. Currently, brand-name commodity dual CPU servers are commonly available for less than US$2,000, with well-equipped, top-of-the-line versions costing less than US$4,000. Based on these prices, it is possible to achieve the same number of millions of instructions per second (MIPS) as a certain server that has more than 70 CPUs for less than US$50,000 by using commodity servers—a savings of almost US$3.5 million, or 98.5 percent.

The amount of compute power that can be inexpensively assembled by use of commodity hardware is staggering. However, the compute power is worthless without the means to address the actual problems at hand.

For the first tier—the Web tier, which consists of a farm of Web servers—commodity servers can be incredibly effective. Each of these servers is typically responsible for managing client connections, parsing HTTP headers for sticky load balancing information, and even handling secure sockets layer (SSL) encryption and decryption duties, which are very computationally expensive. Because each server has its own dedicated input/output subsystem and because the servers do not have to communicate among themselves to handle their load, they provide an extremely cost-effective solution that exhibits linear scalability for the Web tier.

In the second tier, which is often an application server cluster, commodity servers can provide very good scalability for applications that are generally stateless. Exceptions to this are usually related to HTTP session management. Most

application servers support sticky load balancing from the Web tier, which means that a particular HTTP session can be managed locally by one machine as well as having options for HTTP session failover and migration on demand —which can be necessary when the sticky load balancing doesn't stick.

Because the database tier is responsible for managing application data in a fully resilient manner, it also presents the most complications for implementing a scale-out architecture—a topic that is well beyond the scope of this paper. Suffice it to say that advances in products such as Oracle Real Application Clusters (Oracle RAC) are making it possible to implement a scale-out architecture for the database tier.

All three of these scale-out architectures are loosely referred to as clustering, although they differ significantly. In the case of the Web tier, no server-to-server communication is even necessary, whereas at the other extreme, the database tier must manage transactional consistency across its cluster.

This white paper focuses on the application tier. Because that tier is responsible for the bulk of the application logic, it tends to be responsible for the dynamic nature of the application, the page generation, the user session state, and many responsibilities that unfortunately cannot be pushed forward into the Web tier. Further, if the application tier allows its operations to become requests back to the database tier, the impact on scalability will be devastating. As such, solving large-scale data access requirements in the application tier without delegating the entire load to the database server is a key step toward achieving scalable performance.

CLUSTERED CACHING

Clustered caching refers to the ability to maintain data in the application tier in such a way that the application can fulfill some portion of its data access requirements from the cache.

Clustered caching refers to the ability to maintain data in the application tier in such a way that the application can fulfill some portion of its data access requirements from the cache. This mitigates the application's load on the database without violating the application's requirements for data correctness if that data is being changed.

Traditional caching is based on two fundamental concepts: The first reflects a limitation on the size of the cache, because the cache must be prevented from growing so much that it negatively affects the application that is maintaining it (for example, by causing the application to run out of memory). The second involves the application's degree of tolerance for out-of-date (stale) data. The risk of stale data arises as soon as an application chooses to hold onto data instead of relying on the system of record for the data. When an application reads from a traditional cache, it is accepting that the data might be stale. By discarding data from the cache once it has been cached for a certain period of time, the cache expires potentially stale data. In other words, the application is assured that its cache will not provide data that is stale for a period longer than what is specified by its cache expiry setting.

Even in a clustered environment, this simple traditional caching approach can be very effective for frequently requested data when the application can accept a certain amount of staleness.

For distributed systems, an extension to the traditional caching approach is for a server to notify other interested servers when that server has modified a particular piece of data. This reduces the likelihood of the cache's containing stale data in an environment where multiple servers have a copy of that data in their own cache, because those servers will evict stale data from the cache as soon as they are notified that it has been changed. Again, this traditional distributed caching approach can be very effective.

Coherent clustered caching is the ability of servers within the cluster to know that the data in the cache is up-to-date.

Coherent clustered caching is the ability of servers within the cluster to know that the data in the cache is up-to-date, and provides the application with the means to totally eliminate the possibility of cache staleness when necessary without transferring additional load to the database. Coherent clustered caching implies the ability to synchronize on the cache, or elements within the cache, in much the same way that application logic can synchronize on an object to guarantee a thread-safe implementation. Furthermore, some caches provide transactional semantics, enabling an application to modify the caches and accept or reject all the changes to the cache in a manner that maintains transactional consistency.

Implementations of traditional caching and distributed cache invalidations are common and well understood; the concept of maintaining cache coherency in a cluster is much more novel and much less understood. Because it can eliminate the potential for stale data and the resulting concerns, however, application architects much more readily embrace coherent clustered caching as an acceptable caching solution for scale-out application environments.

CACHE REPLICATION

The best-known form of coherent clustered caching is the fully replicated cache. Replication itself is not responsible for synchronous coherency, because it would conceivably have the same small communication time windows that would allow for staleness. Rather, it is the ability to achieve guaranteed coherency by synchronizing against the cache or its constituent data or by modifying the cache within a transaction.

Unlike invalidation, which throws away the data the application is using, cache replication keeps the data as it has been modified and shares that modification with the other members of the cluster. As such, it is a "push" model, because it pushes new data to the other servers in the cluster as soon as the data is available.

The purpose of a replicated cache is quite obvious. If each server maintains a local copy of cached data, then the application logic running on each server can access local data without the need to communicate with any other servers. As a result, data access has no measurable latency. Figure 1 illustrates this purpose:

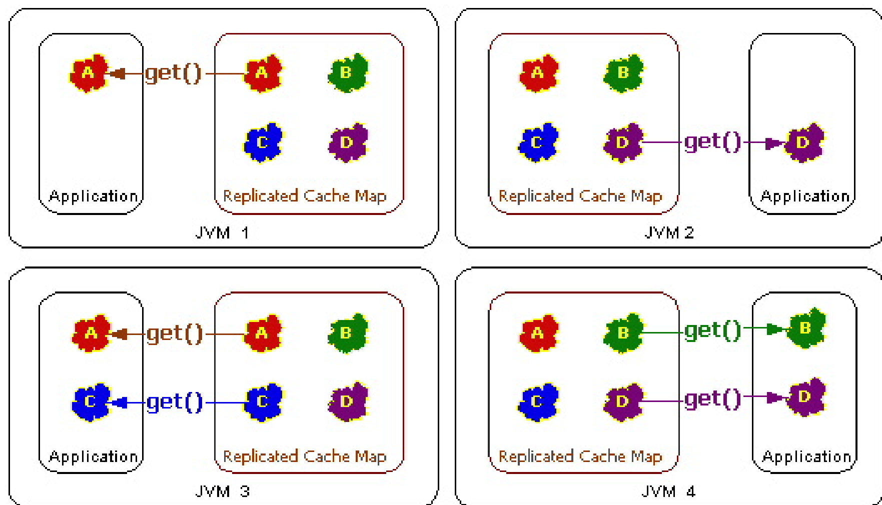

Figure 1: Four application servers accessing data from a replicated cache

Just as the purpose of a replicated cache is obvious, the limitations are equally obvious. First, maintaining the cache across the cluster when a change occurs implies the need to communicate to the rest of the entire cluster. Such communication—often accomplished by use of group network protocols—cannot by its nature scale linearly. Second, the cache is severely limited in its in-memory size, because each application server is maintaining the entire cache within its process space.

CACHE PARTITIONING

To solve the limitations of a replicated cache model without sacrificing either the high-availability (HA) benefits of redundancy or the coherency guarantees provided by the clustered cache, Oracle invented the concept of a shared-nothing cache architecture, called partitioned caching, shown in Figure 2.

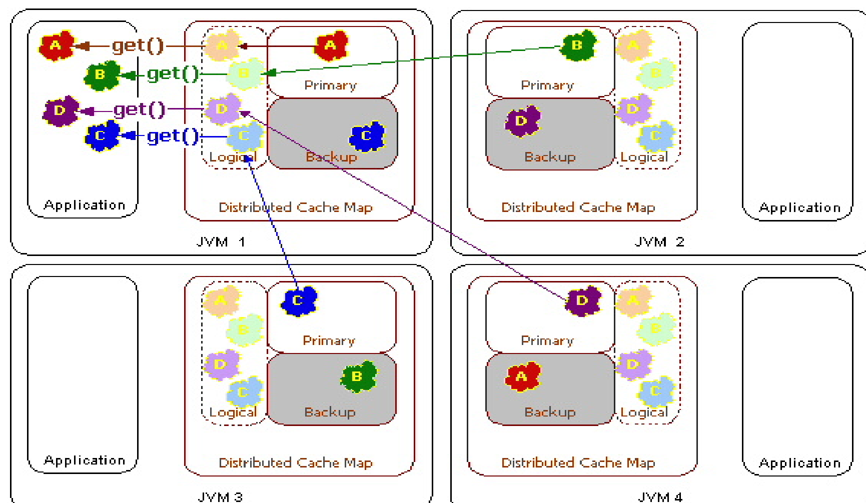

Figure 2: The same application accessing the same data from a partitioned cache

With the data partitioned, the cache capacity grows linearly with the size of the cluster, as does the processing capacity available for managing the cache. Further, in a shared-nothing architecture, each piece of data in the cache has exactly one owner within the cluster who is responsible for managing that data. (In Figure 2, this is designated as the "Primary"; the "Backup" is a synchronous copy maintained solely for failover purposes.) All network communication can be point-to-point in a partitioned model, allowing the cache throughput to scale linearly on a switched network.

NEAR CACHING

Partitioned caching introduces two fundamental latency issues: One is the obvious network time involved in obtaining the necessary data from the server that owns it. The other is a process known as deserialization that must occur to turn the raw binary data back into an object. This process tends to contribute more to latency than the network time.

Partitioned caching introduces two fundamental latency issues: One is the obvious network time involved in obtaining the necessary data from the server that owns it. The other is a process known as deserialization that must occur to turn the raw binary data back into an object. This process tends to contribute more to latency than the network time.

To resolve both of these latency issues, a near cache is used to "cache the cache." Once the network time and deserialization are complete, a copy of the resulting object is managed locally to avoid the need to repeatedly obtain it across the network and deserialize it.

Figure 3 illustrates this and shows how the partitioned cache can be used as an entirely out-of-process cache by the application. The application is able to obtain the data from the partitioned cache running on *cache servers* (the lower two servers), whereas the application servers use a near cache to eliminate the associated latency.

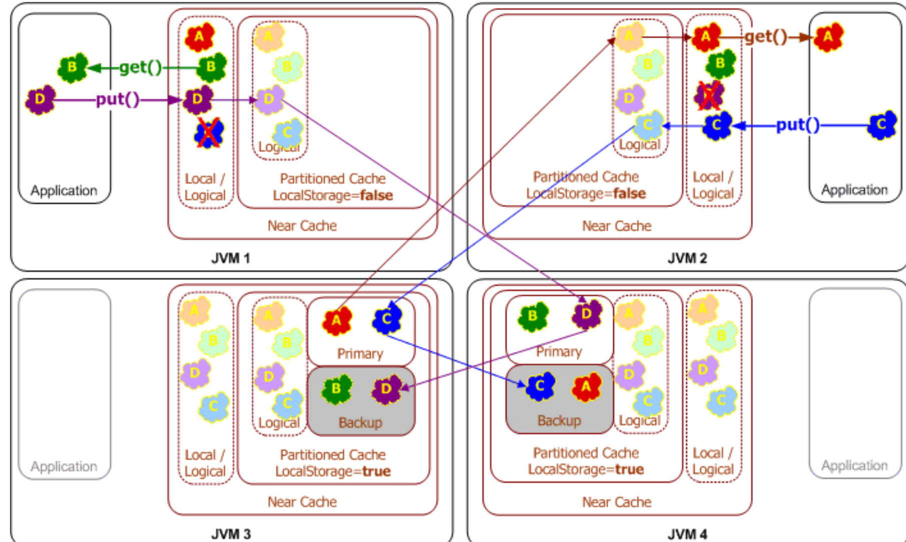

Figure 3: Same application, same data, with a near cache of a partitioned cache

Figure 3 also shows an invalidation pattern being used to keep the near caches in sync, as illustrated by the invalidation-based eviction's occurring in the first server.

CACHE-THROUGH ARCHITECTURES

Cache-aside is a common approach to caching in which the application is responsible for obtaining the data from the datasource if the desired data is not in the cache. And having obtained the data from the datasource, the application will place the data into the cache so that subsequent accesses for the same data can benefit from the cache. This approach is often used to add caching to an existing application and involves adding a cache access immediately before the datasource access, by conditionally short-circuiting the datasource access if there is a cache hit and by adding a cache update after the datasource access.

Cache-through places the cache between the client of the datasource and the datasource itself, requiring access to the datasource to go *through* the cache. When the application needs data that could be cached, it simply requests that data from the cache and the cache, in turn, returns the requested data. The process that occurs within the cache to provide the data is conceptually similar to that of the cache-aside model, except that the responsibility for accessing the datasource has now been pushed down to the level of the cache itself. The cache accomplishes this by delegating the responsibility for the actual data access to a *Cache Loader* implementation (see Figure 4).

***Cache-through* places the cache between the client of the datasource and the datasource itself, requiring access to the datasource to go *through* the cache. When the application needs data that could be cached, it simply requests that data from the cache and the cache, in turn, returns the requested data.**

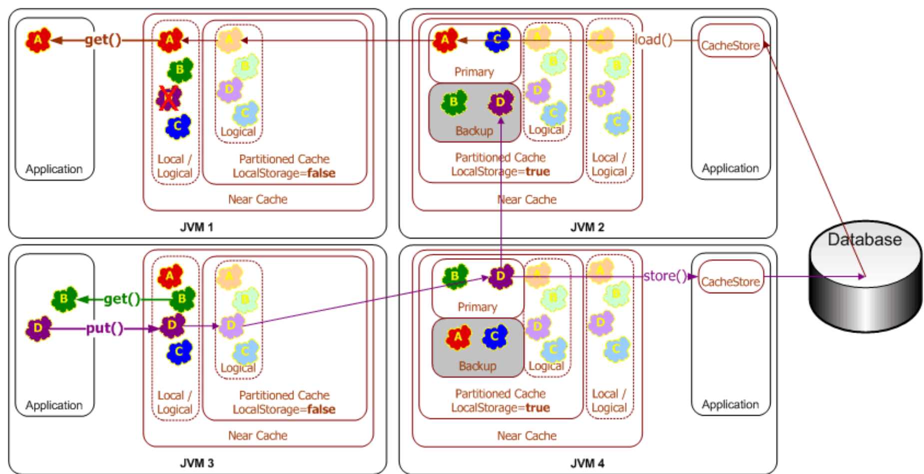

Figure 4: Same application, same data, using read-through/write-through

The partitioned approach guarantees an owner for each piece of data in the clustered cache, so if the application requests a piece of data that has not yet been loaded (or that has been evicted), the cache can load that data by using a cache loader. Because the partitioned approach has the ability to serialize access to a particular cache element, it can handle numerous concurrent requests for the same data with a single load, which helps significantly reduce the load on the underlying database.

REFRESH-AHEAD AND WRITE-BEHIND OPTIMIZATIONS

Refresh-ahead caching is used to preload data that is soon to expire from the cache. The latency of the corresponding database operation is then eliminated from the

user experience, because the refresh can complete before the data expires. Although refresh-ahead caching may not itself reduce load on the database, it will improve the experience of end users, by making sure the data the end users will be using will be waiting for them, in the cache and up-to-date.

Using *write-behind* caching is another way to eliminate latency. Unlike the write-through architecture, which makes sure the database accepts the data before committing the same data into the cache, the write-behind architecture writes it directly into the cache and triggers an asynchronous—and potentially deferred—write to the database. With asynchronous writes, the latencies of database updates are eliminated from the end-user experience (see Figure 5).

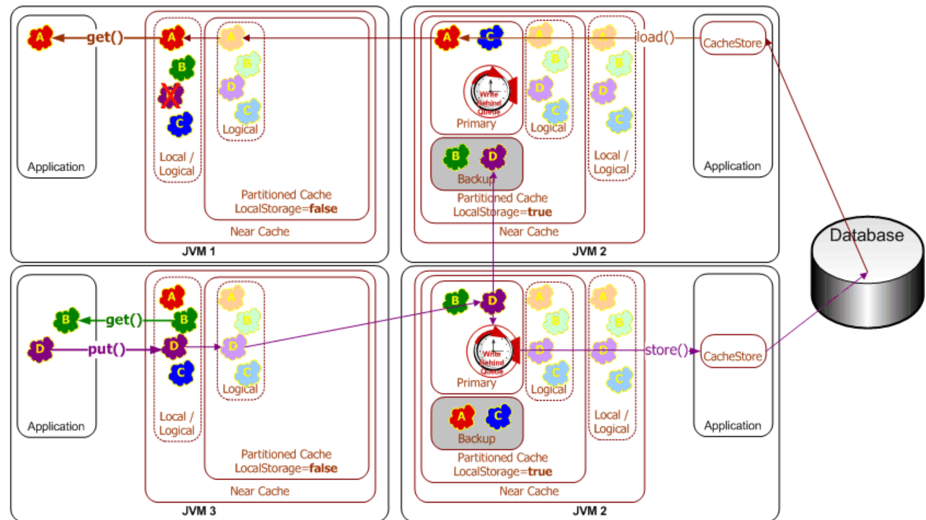

Figure 5: Same application, same data, using write-behind caching

Furthermore, if multiple changes are made to the same cached data while it is in the write-deferred period of time, only the most recent data will be written back to the database when the write-behind caching actually occurs. The result can be dramatically decreased load on the database, meaning much better responsiveness for database transactions that must be performed synchronously.

IMPLICATIONS FOR WEB SERVICES AND SERVICE-ORIENTED ARCHITECTURES

Service-oriented architectures (SOAs) and Web services in general exhibit the same requirements for scalable performance as any other line-of-business or outward-facing application. Much as advanced Web applications manage HTTP sessions to provide conversational states on the server, Web services often have to implement stateful conversations. In fact, they are sometimes implemented by use of HTTP sessions. On a request-by-request basis, the data access requirements for Web services appear to be significantly higher than for Web applications, due to the nature of Web services, in which ancillary data is often included in a response to eliminate the need for subsequent requests. In some cases, the request volumes are

also significantly higher and growing at a much higher rate, largely because the "clients" are no longer humans impeded by considerations such as "think time."

For these reasons and more, the adoption of clustered caching for Web services architectures is occurring both sooner in the technology cycle and at a significantly greater rate than that observed with Web applications.

CONCLUSION

As this white paper has demonstrated, clustered caching is a proven approach that can provide applications with significantly higher throughput and lower latency for data operations while retaining the appropriate levels of data quality that the applications require. The result can be greatly improved scalability.

Achieving the Impossible: Unlimited Application Scalability
Updated May 2007

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2007, Oracle. All rights reserved.
This document is provided for information purposes only and the
contents hereof are subject to change without notice.
This document is not warranted to be error-free, nor subject to any
other warranties or conditions, whether expressed orally or implied
in law, including implied warranties and conditions of merchantability
or fitness for a particular purpose. We specifically disclaim any
liability with respect to this document and no contractual obligations
are formed either directly or indirectly by this document. This document
may not be reproduced or transmitted in any form or by any means,
electronic or mechanical, for any purpose, without our prior written permission.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.