High Speed Data Caching at Barclays Capital

Stephen J. Mueller and Dillon J. Buchanan

December 18, 2009

Abstract

We designed and developed a benchmarking framework to be used by Barclays Capital Inc. to measure the performance of Oracle Coherence. Next we gathered and analyzed important data, including query, update, and load response times, fail-over performance, and latency and throughput of replication across Wide Area Network with a cluster in London. Ultimately, we tested numerous configurations and strategies to determine the optimal configuration and provided recommendations to the firm to achieve high performance. We concluded that Coherence is a scalable and fail-safe system that would perform well in a production environment.

Contents

List of Figures

List of Tables

Chapter 1

Executive Summary

Barclays, like many other companies that have a history of acquisitions, has a number of different storage systems that are currently used in production. Data in different groups is separated, and downstream applications are required to access multiple sources of data. There is an initiative within Barclays to reorganize the way data is stored and aggregate all data into a centralized system. The goal is to reduce the amount of data replication and improve the performance of downstream applications by improving the time data requests are serviced in.

Within the new initiative, plans had been laid out intending to use a cache based on Oracle Coherence to improve the response time of frequently accessed data and to assist the database in servicing requests. This project tests the performance of the Oracle Coherence cache as it would be used by Barclays in production.

The first objective of this project was to create a benchmarking framework to test the performance of the cache. This was used throughout our project for testing and can be used in the future to test the performance of Coherence with other data objects.

Our other two core objectives were to run the benchmarking suite against the Oracle Coherence cache to determine optimal caching strategies and to quantitatively determine what type of performance could be expected from the Oracle Coherence cache if it were to run live in production. We specifically tested strategies for loading data into the cache,

updating the cache, querying the cache, and JVM strategies.

Regarding loading data into the cache, there were three key tests that we ran. The first test was to determine the amount of time it would take to populate the cache based on loading the cache from a database using Java Database Connectivity (JDBC). Loading using JDBC is a standard procedure, and involves creating an explicit mapping between rows in the database and variables in the Java object. We found that the system took fifty seconds per million Position objects and 112 seconds per million Activity objects to load the cache using JDBC.

The second method that we tested was the Java Enterprise Database Integration (JEDI), a proprietary solution developed by Barclays for abstracting away much of the code for connecting to a database. This performed quite well comparatively, requiring fifty-four seconds per million Position objects and 121 seconds per million Activity objects. The advantage to using this method is that it shortens the development phase by reducing the amount of code that needs to be written.

The next strategies that we tested were update strategies. Barclays determined that it is likely that they will need to update 40% of the data in each cache after the market closes. The two methods tested were to create a new cache and destroy the old one, and to overwrite 40% of the data with new data. The latter method proved to be faster requiring 140 seconds, and 341 seconds to perform the former.

When testing queries, we chose eight queries that were likely to be run in production and captured baseline numbers for them. We noticed wide variation in the response times, which we realized was heavily influenced by the amount of objects returned. We also determined that running a query against three caches performed on average 17% faster when run in parallel than when run in series.

We realized that most of the data fields were represented as Strings. We devised a test to run the same query based on an Integer and a String. Comparing the results showed us that you can obtain a 23% performance improvement by changing the data type from a String to an Integer. For this reason, we recommend representing heavily queried upon fields, such as date, as Integers if possible.

It also became apparent to us that queries based on a primary key had far better performance than all other queries. Other queries typically required between 400ms and 10s to complete, depending on a number of factors, whereas queries based on the primary key were completed in 1ms. This is a significant improvement in performance, and we would recommend analyzing the possibilities for developing a system to take advantage of this. We propose one such solution in 8.5.

After completion of the project's core objectives, we approached the time permitting objectives: to benchmark the performance of replication across the Wide Area Network (WAN) and to integrate Coherence with a .NET client and test the performance of "near cache".

Replicating the New York cache across the WAN to a Coherence cluster in London provided a new set of challenges. Initially, replicating the cache caused load time to increase forty-two times while using Coherence Push Replication 2.3. We found that in push Replication 2.4 and 2.5 there is an ability to specify a write delay to the publishing buffer. This significantly improved performance, and with write-behind and Push Replication 2.5, loading with replication requires 60% more time than without.

We also noticed that when using Push Replication 2.5 we learned that it is important to have multiple proxy servers and multiple threads per proxy server in order to handle the increase throughput.

Our final objective was to integrate Oracle Coherence with a .NET client. We wrote a client using .NET that could perform queries against the Coherence cache. We found that query times between a Java client and a .NET client were similar, with response times within 10% of each other.

The results of our testing led to many recommendations, which are covered in 6. The remainder of this report covers more detailed information that can be used by Barclays to make more informed business decisions about possible future investments in an Oracle Coherence production cache.

Chapter 2

Introduction

Today's economy relies heavily on the transportation of electronic data. Everything from equity transactions to ATM withdrawals are accomplished via computer systems. Banks and other financial institutions are now embracing the technological advances made in the areas of Computer Science and Computer Engineering more than ever. Performance and scalability have become the cornerstones for financial data storage centers. Thus, it is to the institution's advantage to create the most efficient and scalable transaction system feasible.

Barclays, a worldwide financial service provider, currently wishes to upgrade their transaction architecture. Currently, price-point transactions are constantly aggregated in individual data locations. At the end of every business day, data from each server is consolidated and published for utilization by various financial applications. The trouble with this current architecture is that there are multiple sources for data, and downstream applications often need to connect to many sources of data to get the information they need. Because the same data may be present across multiple individual servers it requires constant maintenance to assure that the data is synchronized properly. In addition, because each server acts independently, it may become a point of resource bottle-necking. Finally, the fact that the current architecture must wait to the end of the day to consolidate data means that sudden opportunities may be missed and only reviewed at the end of the day. A solution to this problem is a unified data storage system, consisting of a single system built on a persistent data source and and a caching system to assist with frequent requests.

A package called Oracle Coherence offers possibilities by providing a distributed caching application over a grid of servers. Coherence consolidates and administers the resources and processing power of each server in the grid. This essentially creates a synchronized data environment with load-balancing capabilities. Information can be accessed rapidly because there is no single bottleneck in the system. Coherence transparently replicates data across all available servers so applications access the same data no matter which server they access. Also, in the case of a server failure, the workload is redistributed seamlessly so that all applications can continue without failure.

This project consisted of three core objectives: to develop a benchmarking framework to test Oracle Coherence for the type of applications that Barclays wishes to use it for, to determine optimal strategies to get the best performance out of Coherence when performing integral business tasks, and to quantitatively determine the type of performance to expect from the Coherence cache. We developed a framework that allowed us to benchmark the performance of the cache using two objects specified by Barclays, and also made the framework extensible so that it can be used in the future to benchmark the performance with other data objects. Through testing, we were able to determine optimal strategies for performing loads, updates, and queries on the Oracle Coherence cache to help improve end performance. We also recorded quantitative measurements for each test, correlated it, and provided it to Barclays so that those within the company will be able to make informed investment decisions regarding the viability of an Oracle Coherence cache.

The project also contained two bonus objectives to be completed if time permitted. We completed the first of these by testing and gathering strategies for replicating the cache in New York City with another Coherence cluster in London. The second of these was to test Coherence's integration with .NET and to compare the performance improvement of using a "near cache". We were also able to complete this objective in the allotted time.

This project consisted of a number of key objectives. We were able to quantify the type of performance to expect from an Oracle Coherence cache, demonstrate its ability to replicate data across the WAN, and integrate it with a .NET client.

Chapter 3

Background

Before starting the project, it was necessary to familiarize ourselves with the topics at hand. While many of the topics were custom to Barclays situation a majority of the rest were subjects dealing with programming applications. Understanding the material below provided the foundation which our project was built upon.

3.1 The Live Environment

In order to appropriately benchmark the expected performance of Oracle Coherence in production, we needed to understand the production environment. The figure below, generated from prior statistics of the current system, depicts the load on the server against the time of day. There are several key features of this graph. The first is the rise at the beginning of the day. This is due to the influx of trade requests during a closed market the previous day. As requests are fulfilled throughout the morning the load against the server begins to dissipate. During the middle of the day the server sees a period of reduced load. This, however, is met with another rise in load due to the future closing bell. As traders rush to buy or sell, the activity on the server begins to spike as it did in the morning. This continues up to the closing of the work day. Finally, during sometime in the night, the database and cache rectify any discrepancies between them. Thus, it causes a huge amount of activity on the server for a brief period of time till the synchronization is complete.

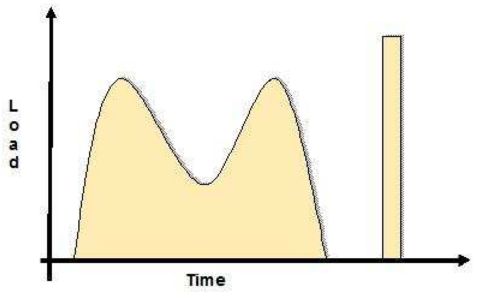

Figure 3.1: Server Activity (Load) vs. Time of Day

As we will show later, we tried to mimic this type of load through our own load generation. We do feel that it would be useful to quantify, in terms of frequency or throughput of queries and updates, what the peaks of that graph would be, however this was beyond our capabilities.

3.2 Current Setup

The Coherence Cache that we are implementing is a part of a larger initiative inside Barclays to change the way data is stored, called The Hub. In the current setup, data from the front office is pushed to many different data storage systems. On the other end, there are many different applications that pull from multiple data sources. While the specifics of this are confidential, the following diagram is an approximation of what it looks like.

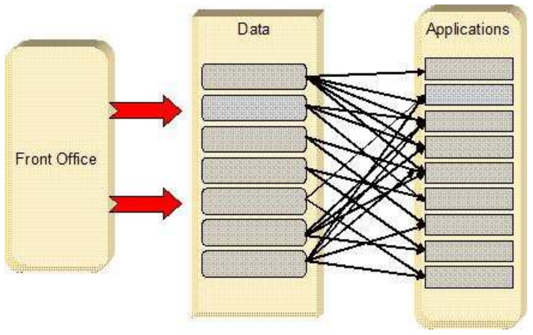

Figure 3.2: Current Architecture

The fault of the current architecture lies within the “Data” boundary. As Barclays grew, more storage resources were allocated and appended to the “Data” system vertical. Each node within this vertical is independent of every other node and only persists data pertinent to itself. The problem then arises when applications from subsystems attempt to access information across multiple nodes. For an application requiring information on positions and activities multiple requests would have to be made to multiple data sources. This is symbolized by the crossing black lines between the “Data” vertical and the “Applications” vertical. As you can see by all of the crossing lines, there is a general lack of organization in the way this is done. It was probably not designed to behave like this, but is the result of many different business segments coming together.

To demonstrate the inefficiencies of this architecture let us say that the end application is a tool used by a bond trader to check different financial and bond data. This program may need to connect to multiple sources of data, one for trades that were placed in the US, one for trades placed in the UK, and one for different sources of economic data. Latency from any of these data sources could cause the end application to slow down. This delays the applications response time and may upset any customer whom is patiently waiting on the result. Also, it has been mentioned that there is often some duplication of data between the data sources, calling the quality of the data into question if an update was not performed on all instances of a particular piece of data.

A re-organization of this system is imperative. A more centralized and organized data system would allow Barclays to provide quicker responses to internal departments as well as better service to their clients.

3.3 Oracle Coherence

Oracle Coherence is an in-memory distributed data grid software solution for clustered applications and servers. Coherence provides replicated and distributed data management services on top of a reliable, highly scalable peer-to-peer clustering interface. [9]

Oracle Coherence excels at consolidating and managing mission-critical data in extremely responsive media types so that client applications may have access to the data with as little latency as possible. This methodology allows the cluster to pool its resources and essentially create one large data repository. Data that is used frequently may be stored in local caches while data that is seldom used may be stored in back-end storage mediums.

Coherence has no single points of failure; it automatically and transparently fails over and redistributes its clustered data management services when a server becomes inoperative or is disconnected from the network. When a new server is added, or when a failed server is restarted, it automatically joins the cluster and Coherence fails back services to it, transparently redistributing the cluster load.

Organizations can predictably scale mission-critical applications by using Oracle Coherence to provide fast and reliable access to frequently used data. Oracle Coherence enables customers to push data closer to the application for faster access and greater resource utilization. By automatically and dynamically partitioning data in memory across multiple servers, Oracle Coherence enables continuous data availability and transactional integrity, even in the event of a server failure. Oracle Coherence is a shared infrastructure that combines data locality with local processing power to perform real-time data analysis, in-memory grid computations, and parallel transaction and event processing.

3.4 Proposed Solution

The new initiative inside Barclays aims to redesign how data is stored, combining data across different business units and different verticals. The new system will push all data from the front office to a more centralized data storage system that all end applications will pull from. This will create a single source for data based on a standardization of technology. The intention is that data requests will be performed more accurately and quickly, thus improving the quality of service to Barclays' customers.

The graphic for the proposed solution is again confidential, however, the graphic below demonstrates what this new system will look like from the perspective of this project.

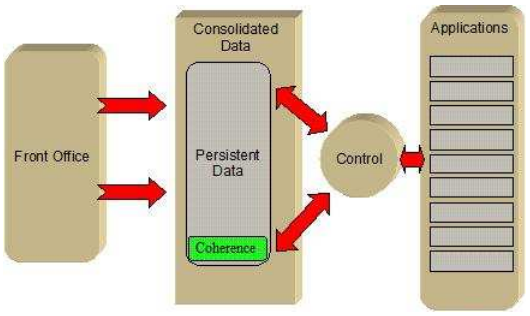

Figure 3.3: Proposed Architecture

In the setup above, the consolidated data is stored in the form of persistent data, and the Oracle Coherence cache. When an application is attempting to access data, the control block will determine whether the information will be stored in cache (recent data) or whether the data will need to be retrieved from the persistent data source. We are benchmarking the performance of the Oracle Coherence cache, the high speed component of the new data storage system.

3.5 Testing Environment

We ran our initial tests on four different servers, each containing six instances of Oracle Coherence, running on a separate Java Virtual Machine with 2GB of heap space each. This gave us a total of twenty-four nodes, and a total of 48GB of memory to store our caches. The server configuration can be seen in the diagram below.

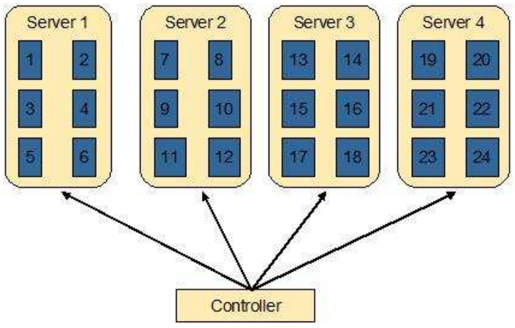

Figure 3.4: Physical Setup

As you can see above, we also have a controlling machine. This is the machine where we launch our test scripts from and where we gather all our logs. In addition, the controller box allows a centralized way to control the services being executed on all the child servers. This provides the ability to start, stop, and configure every child server from one centralized machine. On top of configuration, this machine is tasked with running our performance tests against the data cluster.

The above diagram shows the physical setup for our tests, however, this is independent of the logical setup of our caches. Our initial configuration included eighteen different caches; nine for Position data and nine for Activity data. This is represented in the table below.

	Positions			Activities		
	T	T-1	T-2	T	T-1	T-2
NY	P_NY_T	P_NY_T-1	P_NY_T-2	A_NY_T	A_NY_T-1	A_NY_T-2
EU	P_EU_T	P_EU_T-1	P_EU_T-2	A_EU_T	A_EU_T-1	A_EU_T-2
ASIA	P_ASIA_T	P_ASIA_T-1	P_ASIA_T-2	A_ASIA_T	A_ASIA_T-1	A_ASIA_T-2

Table 3.1: Logical Setup

In our initial setup, we have chosen to cache the previous three days worth of data (labeled "T", "T-1", and "T-2"). We are also caching data for three different regions, the United States, Europe, and Asia. We expect to get better performance from smaller caches, so we started with an initial setup where each combination of time and region was a separate cache for both activities and positions.

It is important to note that the physical setup, with twenty four nodes on four physical machines, and the logical setup, with eighteen different caches, are independent. The actual location of data storage is hidden from the developer that uses Oracle Coherence; we do not know which node a cache will be stored on, or even that a cache will be stored at a single node. Coherence also has redundancy in the data, also hidden from the developer, so that if one node fails, they data can be recovered elsewhere.

3.6 Data Objects

In our tests, there were two data objects that were used. The group that we are working with at Barclays is interested in tracking data for Positions and Activities (or trades). In a database, these objects are stored in a table, with each attribute corresponding to a row. In Oracle Coherence, these are stored as actual objects, where each attribute is stored in an instance variable. For our purposes, we created classes called PositionType and ActivityType, which were simple classes that had the necessary instance variables, along with getter and setter functions. As demonstrated later, the data types found within these objects are in direct correlation those of the database. This allows for a direct translation between the objects in the database to the objects that are inserted into the Coherence cache.

3.7 Programming Tools

The subsections that follow describe several of the programming tools we used to ease production and accelerate development of our Oracle Coherence benchmarking suite.

3.7.1 JavaBeans and Spring

Spring is a programming framework that can be used to make a Java application more flexible. The Spring framework provides a specification for an XML configuration document that allows a user to specify the attributes of a JavaBean, which can be used in the Java application. This feature was used extensively in our project, as it made our code more reusable and it made it easier to run different tests without re-coding and re-building large portions of the project. A JavaBean, as defined by the JavaBeans white paper, is “a reusable software component that can be manipulated visually in a builder tool”. This leads to two main requirements for a JavaBean: it must have simple “getter” and “setter” functions that are exposed as public so that it can be manipulated, and it must not tie in specialized libraries so that it can be highly re-usable. This simplicity allows a programmer a clean way of separating object attributes from the compiled code. The power of this can be explained best by an example, shown below. [3]

```
<bean id="Query" class="com.barcap.hub.request.query.ActivityTradeDate">
  <property name="day">
    <value>Jan 24 2009</value>
  </property>
</bean>
```

Figure 3.5: JavaBean Sample Code

The XML above specifies a bean called "Query" which is of the type "ActivityTradeDate", one of the classes we wrote to query the cache based on a specific date that a trade was placed on. Our benchmarking framework will look to instantiate a Query named "Query" from this configuration file, and will then run it. If we would like to change the query that gets run, or the date that this query is run on, we do not need to recompile the project or repackage the jar, we simply change the Spring configuration. This concept saved us time during our testing, and will make our code easier to re-use in the future.

3.7.2 JMeter

JMeter is an open source Java project that can be used to run performance tests on a wide variety of applications, from websites, to Servlets, to Java Objects. It has a Graphical User Interface (GUI), which allows for the setup and configuration of tests, looping functionality, and the display of results. JMeter allows you to create test plans, where you can invoke certain parts of your application. It can be run and controlled locally, controlled locally and run remotely, or the test plan can be exported to a file and then it can be run directly on the server you are testing, in which case all data gets stored to an XML log file.[4]

For our project, we relied on JMeter to be the engine behind our test plan, and use it to invoke our Query classes, Update classes, and Load classes. From the JMeter GUI, shown below, we are able to configure many of the input parameters that we want our test to run with. We then export the test file and place it on our server, and are able to run the test from there.

Building and testing the test plan through the GUI and then running it from the command line on the server combines the flexibility and ease of use of the GUI with the speed and performance of executing it from the command line.

JMeter also has the capability of running multiple threads, which allowed us to test queries that go to multiple caches, and many queries or updates running at the same time.

Combined, JMeter and Spring have made our benchmarking suite flexible and easy to use, making it easier for ourselves and other to test various situations and server configurations.

Chapter 4

Methodology

In order to fulfill each of the objectives for the project, there were a number of key design steps that we needed to take. First, we determined the important metrics for measuring a data cache and rated them in terms of relative importance. We then compared this to all of the possible Oracle Coherence cache configurations, and eliminated any of the configurations that would not meet the previously gathered requirements. After eliminating many of the cache configurations, we were then able to proceed in creating a benchmarking framework to test the cache and determine optimal strategies for performing key business functions. Careful planning and use of software engineering principles were required during the design phase of our benchmarking framework in order to make it easily extensible in the future. This saved us time later on, making it easier for us to reach our bonus objectives, and will allow Barclays to use the framework to test other data objects.

While designing the benchmarking framework, we integrated it with other existing technologies such as JMeter in order to leverage their capabilities and help us perform our tests quickly and efficiently. We also designed a system of shell scripts to run all of our tests and external configuration files to ease in changing the test parameters. In addition, we used the benchmarking framework to test the performance of the cache to determine strategies for updating, loading, and querying the cache. With the completion of our core objectives, it was then necessary for us to determine and design methods to test performance of replicating the cache across the WAN with London. Finally, we created and ran tests that integrated .NET clients with the Coherence cache and tested the performance of the "near-cache distributed" caching configuration.

4.1 Metrics

Our first task was determining what metrics in this project were a high priority and which were not. Working with the Barclays team, we constructed the metric table below that depicts the importance of several project variables.

Metric	Description	Importance
Update / Insert Latency	High speed data transfer allows for timely decisions.	High
Read / Write time to long term data storage	After certain period of time, data will be transferred to persistent memory. Don't want this to degrade system performance.	Low
CPU Utilization	Indicates whether the system could handle a greater load.	Medium
Memory Consumption	Indicates whether the system could handle a larger data set.	Medium
Network Activity	High network activity could be a cause for high latency. May already be reflected in latency times.	Low
Garbage Collection Statistics	Critical that garbage collection does not cause interruptions or spikes of high latency.	High
Variance in Response Time	Likely to be caused by garbage collection. Must ensure consistently low response time.	High
Re-balancing Statistics	Ensure that performance stays high while changing load. Ensure that Coherence correctly re-balances load.	Medium
Fail-over Statistics	Ensure that server fail-over proceeds correctly and without data loss. Ensure that performance is still good.	Medium

Table 4.1: Important data metrics and their relative importance.

From the table above, we were able to prioritize our work to coincide with what the bank requested as a deliverable product. High importance items were executed first, while lower priority items were left toward the end of our project's duration. Our final data analysis was based on the measurement of important metrics in each configuration found in the table.

4.2 Setting Up Oracle Coherence

While we were developing the benchmarking suite, we concurrently deployed Oracle Coherence to the server cluster. Oracle Coherence allows for several different cache topologies all of which can be seen in the table below.

	Replicated Cache	Optimistic Cache	Partitioned Cache	Near Cache w/ partitioned cache	Local Cache not clustered
Topology	Replicated	Replicated	Partitioned Cache	Local Caches + Partitioned Cache	Local Cache
Fault Tolerance	Extremely High	Extremely High	Configurable Zero to Extremely High	Configurable Zero to Extremely High	Zero
Read Performance	Instant	Instant	Locally cached: instant Remote: network speed	Locally cached: instant Remote: network speed	Instant
Write Performance	Fast	Fast	Extremely fast	Extremely fast	Instant
Memory Usage (Per JVM)	DataSize	DataSize	DataSize/JVMs x Redundancy	LocalCache + [DataSize / JVMs]	DataSize
Memory Usage (Total)	JVMs x DataSize	JVMs x DataSize	Redundancy x DataSize	[Redundancy x DataSize] + [JVMs x LocalCache]	n/a
Coherency	Fully Coherent	Fully Coherent	Fully Coherent	Fully Coherent	n/a
Locking	Fully Transactional	None	Fully Transactional	Fully Transactional	Fully Transactional
Typical Uses	Metadata	n/a (see Near Cache)	Read-Write Caches	Read-heavy Caches w/ Access Affinity	Local Data

Table 4.2: Different configurations for Oracle Coherence Deployment [6]

Using the metrics we specified in the section above, several of the cache configurations were determined to be unreasonable for this project. The first configuration eliminated was the "Local cache not clustered". Because this configuration had no fault tolerance it was immediately eliminated. Next, because Barclays is operating with millions of data objects within this cache, memory was immediately identified as limiting factor. Two of the cache configurations above rely on the "Replicated" topology which replicates all objects across all servers. This means that the maximum number of items you may have in a given cache cluster may not exceed the minimal amount of memory of any one given machine. This allowed "Replicated Cache" and "Optimistic Cache" to be eliminated from our testing strategy due to their memory inefficiencies.

The remaining cache configurations "Partitioned" and "Near Cache with Partitioned Cache" abide by all our testing metrics and thus, were chosen to be candidates for our cache environment.

4.3 Design Specifications

Prior to construction of our performance utility (described in the following chapter), we addressed the design specifications that needed to be followed in order to produce a logical and concise product. The first task was to conclude how our testing would execute and how it would integrate with our framework. The second issue addressed was how the framework's internal class hierarchy would be constructed. The third topic was was the ability to promote future additions to the framework. Finally, the last specification addressed was which third party technologies would be integrated into the system, and how.

4.3.1 Design Analysis

In the analysis phase of our project we identified several design paradigms that would help us achieve a fully functional performance utility. The first key point identified was how our framework interacted with the various components available. To help us visualize how each portion of our project interacted with each other we developed the following block diagram.

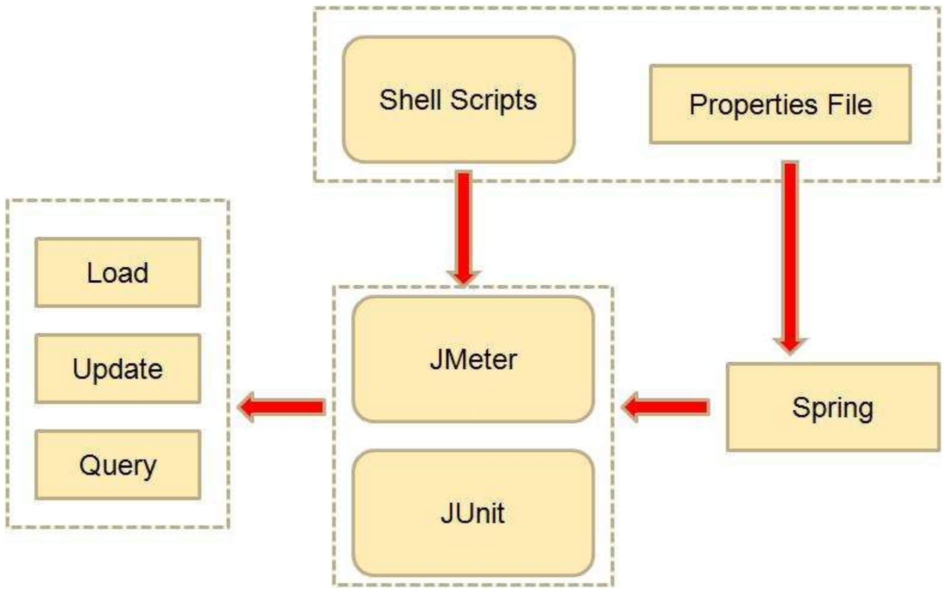

Figure 4.1: Project Components

The figure above identifies several key components within our performance testing utility, each of which are fully explained in their respective section in this paper. The logical flow of progression begins at the top block. This block is where the user directly interacts with our system via shell scripts (4.6.1) and a simple properties file. The properties file is interpreted by the Spring framework (3.7.1) and directly injected into the middle block. Once the user is prepared to execute the system the shell scripts are called which initialize JMeter (3.7.2) and JUnit using the properties passed in by Spring. From here, the JMeter and JUnit systems begin to directly instantiate our Java classes, such as queries, updates, or loads, for testing Oracle Coherence.

From figure 4.1 it was apparent that the collection of blocks in the left portion of the figure would need to be designed specifically for our application. Thus, the next step in our design phase was to develop the Java classes needed to operate our system.

4.3.2 Design Hierarchy

When designing our benchmarking suite, we wanted to use proven software engineering principles. We took advantage of Java's inheritance to ease our development and to enhance re-usability and maintainability of our code.

The chosen hierarchy follows the following UML:

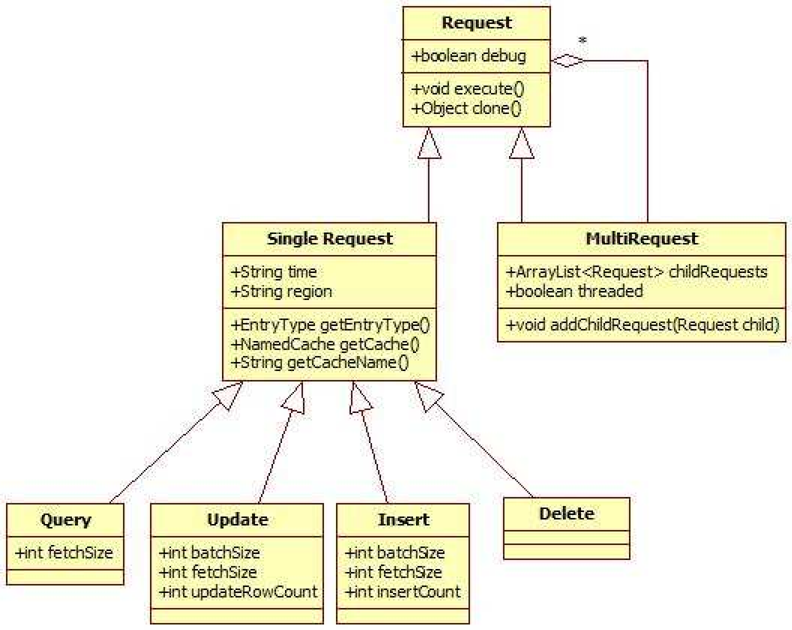

Figure 4.2: Class Hierarchy

As the figure above demonstrates, the base class for all logical operations against the cache system is the “Request” class. This abstract class contains methods and fields common to all requests. From this class, two additional classes are formed: SingleRequest and MultiRequest. The former, SingleRequest, operates on the principal that any sub class will only request information from a single data source and will do so synchronously. The latter, MultiRequest, is a concrete class that allows the user to chain child requests, creating a single request object that may operate against multiple data sources in a parallel manner.

From the “SingleRequest” class, four more specific base classes are constructed, deemed four fundamentals. The four fundamental classes in our project are: query, insert, update, and delete. Each of these classes contained more specific characteristics of their logical names. Finally, on-top of these four fundamental classes stand a multitude of concrete classes. These concrete classes contain even more specific information about the query, load, update, or deletion.

Finally, upon any request execution, the object will return a Response object. This response object contains all subsequent data about the run: affected items, time taken, exceptions throw, etc... In addition, any response has the ability to house child responses from a MultiRequest that spun off a multitude of children. In that case, data is then accessed in a hierarchical manner in which it was executed.

4.3.3 Future Flexibility

The flexibility of our framework was important. Because the project would progress long after we ceased employment it's ease of use and maintainability was vital. To promote ease of use we designed the class hierarchy in such a manner that allowed the user to inherit a simple base class and add any additional logic needed to run the operation.

The user is given a choice of four fundamental base classes: query, update, insert and delete. These base classes already contain much of the logic to initialize a given task, however lack a means of execution. The user is thus required to override the execution method and implement his/her own. This architecture provides the user with all the necessary base information and in return, only requires that the user complete the execution logic.

In addition to the four fundamental classes, the framework allows the user to inherit the base Request class. If a user addition does not fall within the boundaries of the provided fundamental requests, the user is given the ability to deduce his or her own.

4.3.4 Third Party Technologies

When developing our performance testing application it was necessary to incorporate existing frameworks to increase production time. The technologies utilized covered three key sectors: testing, configuration, and database connectivity.

Throughout the construction of our framework it was necessary to design tests to perform regression testing on completed subjects. While a custom test harness was well with the capabilities of design, it was more beneficial to use an already existing framework. Thus, we chose to incorporate JUnit into our framework. JUnit allows the user to write simplistic test cases for all working functions. It then provides a test harness to execute written tests. Using JUnit, we constructed test methods for each class written in our project.

Being able to externally configure our framework saved time so that we did not need to recompile in-between tests. Thus, we chose to incorporate the Spring framework. Spring, as described in great detail in 2.7.1, provides the user with a means of configuring our framework without compiling.

Finally, the last technologies integrated were geared towards accessing remote databases. To accomplish this task we used two sets of frameworks. The first was Java Database Connectivity (JDBC) and the second was an in-house framework called Java Enterprise Database Interface (JEDI). Both were similar in a sense that they allowed remote access to a data base via a simplistic interface, however, JDBC was a proven standard while JEDI was still in its infant stages. While it would have been sufficient to only use JDBC in our framework, it was necessary to also test the performance of JEDI for the benefit of Barclays.

4.4 Extending the Four Fundamentals

The four fundamental requests: query, insert, update, and delete provide the user with a platform class to construct custom requests. These classes, although necessary, are only blank categories for more complex operations to fall under. The true power extends not from the base classes, but from the concrete classes built on-top of these fundamentals. To actually invoke logical operations on the cache, new, more specific classes, were constructed on top of each fundamental.

4.4.1 Queries

Arguably the most important part of the benchmarking suite were the queries. The data gleamed from running the query tests will be most directly related to how the usage of Oracle Coherence will help improve the customer experience, through faster response times.

Each queries below was chosen for a particular reason. The architecture for the cache system is designed to hold two data types: Positions and Activities. Each data type contains over twenty information fields specific to that object type. Thus, to generate queries against all possible fields would have required 40+ classes, each targeting a different field, or a combination of two or more fields. To ease this challenge we chose to implement queries that were most common in a production environment.

Query	Description
Position POF Key	Queries positions based on a custom POF key.
Position Date	Queries positions based on the “business date”.
Position Accounts	Queries positions based on a list of “account IDs”.
Activity Key	Queries activities based on the caches primary key.
Activity Settlement Date	Queries activities based on the “settlement date”.
Activity Trade Date	Queries activities based on a given “trade date”
Activity Trade Status	Queries activities based on the “trade status”
Activity Accounts	Queries activities based on a list of “account IDs”.
Activity Trade Status (3 Time Zones)	Queries activities over three regions based on the “trade status”.
Activity Settlement Date (3 Regions)	Queries activities over three regions based on the “settlement date”.

Table 4.3: Selected Queries

The queries have two interesting qualities. The first is that each are either Activity or Position and the second is that some have an explicit “3 time zones”/”3 Regions” remark. The former accounts for the fact that this system will contain both position and activity information for Barclays’ subsystems. Thus, queries must be designed against both entry type objects. The latter, the time or region remark, or the lack thereof, draws from the concept that there are three distinct regions under test: NY, EU, and ASIA as well as three time zones: T, T-1, T-2. Each time zone and region is a separate cache as demonstrated in section 3.5. The queries that lack the explicit remark are assumed to only query a single region and time. On the contrary, queries that contain the remark, search over all three regions or time zones simultaneously.

One of the most common search methods is by entry date. A client may remember nothing about his or her transaction other than the date it was placed on. Thus, it was extremely important to include this type of query in our testing suite. In addition to dates, account identification numbers were commonly used to query Position and Activity objects. Therefore, both Activities and Positions contained a query against an account ID. Finally, trade status was appended only for Activities, due to the fact that trades that fail require immediate attention to rectify the issue.

The two most promising queries above are the Position POF Key and Activity Key. Coherence's internal storage can be modeled as a hash map. To that extent, it is apparent that the quickest search operation on any hash map is by directly addressing the key of a given value. In fact, any operation against a key completes in approximately O(1). On the other hand, any queries against the values of these keys can be expressed as O(n), n being the number of values in the collection. [1] The value of this is that the cache should perform best when querying a key, rather than a value.

4.4.2 Inserts

The first component of the test suite that we wrote was the loading portion, or insertion of data. The loading component was used in every test that we ran, and it was also one of the insert strategies that we tested. Before testing began, data had to be loaded into all appropriate caches to simulate a live production environment.

To accomplish the loading of external data we chose to utilize an existing database with data objects already present. There are multiple ways for us to accomplish the loading from the database. The first method utilizes JDBC and is the most basic approach. For this method, we load a preset number of rows from the database into our loading application, and go through each column from the database, and call the correct function to assign it to the correct variable. This method is a bit more time consuming to code because the mapping between columns and instance variables must be explicitly written, however, we expected it to be the fastest method because there is little overhead.

The next method that we used to load data into the database was a proprietary Barclays framework specifically created for abstracting away much of the details of interacting with a database. This method does not require an explicit mapping. It uses Java Reflection to parse the code to determine which attributes the Position type or Activity type object has. It then maps the data by matching the names of instance variables with names of the columns in the database. For example, the column "ACCOUNTID" would be mapped to the instance variable “accountid”. Abiding by the naming convention is the only way for this method to work.

The final loading method that we reviewed used JEDI with an explicit Row Mapper. The Row Mapper encapsulates the explicit mapping that we created earlier when using the JDBC. (After creating the Row Mapper, we then used it in our code for the JDBC as well so that we were not duplicating the same code.) This took away much of the advantage of using JEDI, because we still had to create the explicit mapping.

4.4.3 Updates

When testing the update capabilities of Oracle Coherence using our benchmarking strategy, we wanted to test two main approaches: overriding the entire object that was being updated, and using a PofUpdater to target only certain fields that needed to be updated in that object. The former approach required us to pull data from an existing database to replace the object. For this, we again implemented interfaces to the database objects via JDBC, or JEDI as seen in the insert objects above. The latter objective, using the PofUpdater, required no external data source. Instead, this method altered unused data fields of chosen data items. While the alteration of the unused data fields was irrelevant, the goal was to determine the difference in altering a few fields in comparison to replacing the object completely.

4.4.4 Deletes

Constructing the delete classes was quite straight forward. In our case there was only one test case that needed to be covered: deletion of a cache set. Because our data set was stagnant, no individual item needed to be deleted, rather entire caches had to be destroyed. For this operation, we chose to create a class called “Drop”, which borrows its name from Structured Query Language (to drop, or destroy, a table). The class contained a simple Coherence destroy command and returned one affected item, which represented the one cache set that was destroyed.

4.5 Load Generation

While in a development environment it is useful to benchmark a system without any additional load, it is more useful to the business to run tests under the type of conditions that the system will be used in. To this end, we created a load generation application that manifests background traffic by running queries against the cache.

The load generation is started and run in the background so that other testing may proceed while stress is being applied to the system. It accepts command line arguments that specify the duration to run for (in minutes), the number of threads to break of into (each thread gets assigned a query to run), the amount of time to sleep in between each query (in milliseconds), and an optional argument that specifies how often it should try to invoke garbage collection.

The load generation tool picks random time zones and regions to target when each of the threads are created. If a moderately large number of threads (more than 20) is chosen, then this application will be able to simulate the load of actual users, running various queries targeting various caches.

4.6 Execution Strategy

To determine the optimum cache configuration possible a set of execution strategies were devised. Each strategy consisted of a distinct Coherence cache configuration file, as well as a unique performance test suite that would be executed against the cache. The end result of this was a distinct set of configuration permutations. These permutations, when executed, allowed us to quickly and efficiently determine which strategy performed the best and in what sector did it lend performance benefits to (loads, queries, updates, etc. . .). From these results, the best strategies were combined to form a new set of strategies that would again be tested in the attempt to increase overall system performance.

In the following subsections we discuss the tools and processes we've created to increase result throughput.

4.6.1 Shell Scripts

To improve productivity of our environment we chose to utilize the Linux shell script architecture. Shell scripts give the user the ability to perform common command line executions in an automated fashion via a script file. By creating shell scripts we were able to shift the workload of system customization from a manual perspective to an automated one. It thus allowed us to work in parallel while our automated system completed menial tasks that once had to be done manually. Another benefit of this system is that it also allowed us to run overnight regressions that required no user intervention. This provided extremely useful as during any given night several strategies were slated to be executed. By hand, this would require the user to wait till the first strategy was complete, swap out configuration files, and start up the next. However, in an automated environment, this was all done by the machine and required nothing more than a press of the button to begin testing.

4.6.2 Running A Strategy Test

As mentioned previously, the greatest benefit of using shell scripts in our environment was the ability to execute several strategies autonomously during the day. During the morning, when changes are commonly made to the framework, we could easily execute our strategy shell scripts in a "mean-and-lean" fashion by changing a simple input parameter. This meant that loop counters were turned down, fewer rows were loaded into the cache, and other time intensive tasks were minimized. The purpose of this is the ability to quickly test our changes and assure that any new introductions have not adversely affected our benchmark suite.

As the day closed and final commits were made, we then altered our main shell script to begin running with all amenities. This would indicate to the script that all rows were to be loaded, and all loop counters were to be increased to their desired value. The script would then promptly hand off control to our benchmarking suite.

The benchmarking suite, when executed, would immediately begin loading the cache nodes with data. Once data was successfully loaded into the cache, a variety of queries would be executed against the cluster. This provided a baseline for all queries before we began to stress the system. Once the baselines were complete traffic generation would execute. As stated previously, this traffic generation would simulate users randomly querying for data during the day. While the traffic generation was active, another suite of queries would run. This set of queries could then be compared and contrasted with the previous baseline taken before the traffic generation became active. This provided an quantitative way to depict the affect of how other users may drastically affect the performance of every other user in a production environment.

Each test within our test suite concurrently logged all information while it executed. Our framework would immediately log the results to a comma-separated file that would be inspected the next day. Once the framework completed it would exit back to the shell script, and the script would either swap in another configuration file and run the framework again, or quit.

4.7 Selected Strategies

To determine the correct Coherence configuration applicable to Barclays' situation several strategies were conceived. Each strategy created altered the way Coherence operated directly or indirectly. A direct strategy consisted of altering the cache configuration file to change the way Coherence operated internally. On the other hand, an indirect strategy consisted of altering the Java Virtual Machine (JVM) that Coherence ran on. The goal of developing each strategy was to produce a final strategy that optimized Coherence for Barclays' needs.

The table below lists the direct strategies created and their affect on Coherence.

Direct Strategy	Explanation
Default	Baseline for all other strategies to be compared against.
Thread Count	The default service, which all caches are assigned, is provided with additional worker threads.
Unique Services	Each cache was assigned a separate service. Each service is then assigned its own worker thread.
Unique Services & Thread Count	A combination of Thread Count and Unique Services defined above.

Table 4.4: Direct Strategies

The above strategies concentrate on one primary objective: multitasking. The default strategy is included to create a base for all strategies to be compared against. The other three either involve increasing the amount of threads for a default service, creating new services that are each assigned worker threads, or a combination of the two, multiple services each containing multiple threads.

Multitasking is important on a multicored system. To utilize all processing power, multiple tasks are executed in parallel. This is the primary cause for creating the four direct strategies.

The table below lists the indirect strategies conceived to test Oracle Coherence.

Indirect Strategy	Explanation
Sun Microsystem's JVM	This is the most commonly used JVM for all Java applications and serves as a baseline.
Oracle's JRockit	A high performance JVM designed by Oracle
JRockit with GenCon	Oracle's JRockit with the generational concurrency setting enabled.

Table 4.5: Indirect Strategies

These indirect strategies listed above consist of altering the JVM Coherence runs on. Sun's JVM is added to provide a baseline for all other JVMs since it is the most commonly used in all Java applications to date. JRockit, a JVM designed by Oracle, is designed to decrease garbage collection times as well as increase the cached Java code within its internal HotSpot, thus providing better execution times at the cost of more memory. [5] Finally, the last indirect strategy consists of using JRockit with a JVM argument called "GenCon". "GenCon" is short for Generational Concurrency which means that the JVM will attempt to minimize the time spent garbage collecting. The less time spent garbage collecting the more time spent responding to client requests.

4.8 WAN Testing

After completing our core objectives, we started our next objective which was to replicate the data across the Wide Area Network (WAN). It was important to Barclays' business needs that a test of the WAN determine the effects replication would have on the NYC cache in terms of query time and load time, the latency of the WAN, and the throughput that could be loaded through the WAN. For the first of these criteria, we were able to use the benchmarking suite we had already developed. After configuring the system correctly for WAN replication, we were able to run our standard benchmarking suite to determine whether or not the push replication was having adverse affects on the performance of the cache.

In order to measure the latency of the WAN, we needed to design a method to test this that allowed us to capture the time before loading the data object into the New York cache and just after the cache in London receives the data object. Thankfully, all computer clocks inside Barclays are synchronized, so we were able to measure latency by subtracting the time just before the object was put into the cache and just after it was received in London.

To accomplish this, we created a new data object specifically for testing latency. It consisted of a start time, an end time, and a variable length byte array that would were able to use to test the effect object size has on latency. Using this object, we created a special loader to update the start time just before loading the object into the cache. On the London cluster, we created a Passive Listener, a feature of coherence, that would receive the object and capture the end time. Inside this Passive Listener, we would subtract the two times to determine the latency. Logging this result, we were able to perform this test numerous times in succession to determine an average value and a range for the latency. To measure throughput, we used the Passive Listener again. This time, we captured the time the first object was received and time the most recent object was received. Dividing the number of objects received by the total time taken gave us the current average throughput of the system.

In our results section, we show the findings resulting from these latency and throughput tests.

4.9 .NET Testing

Oracle Coherence was originally designed around the Java programming language. However, to engage a larger audience of programmers they decided to replicate their technology into languages such as C++ and the .NET framework. While it was obvious the Java language would be used in Barclays' architecture, it was important from an engineering standpoint, to explore all available options. Thus, we chose to replicate several queries that were originally created in Java over to a sample .NET project. This allowed us to compare both architectures side by side with identical requests to determine the benefits of each architecture.

To accomplish this four queries were taken from our Java framework and recreated in C#.Net: Activity Key, Activity Settlement Date, Activity Trade Status, and Activity Accounts. We then constructed a test harness, in both languages, that would loop each query 10 times and produce the average duration taken to retrieve the correct number of affected items.

The next step required setting up a remote cluster and loading with items. The remote cluster was configured with the default configuration due to the fact that we were not testing the cluster itself, rather the client. Once the loading was complete a proxy server was established on the server side. While Java may naively connect to a cluster, .NET and C++ require a proxy server to communicate with the cluster.

Chapter 5

Results

During our project, we ran a wide array of tests all designed to help increase Barclays' understanding of Oracle Coherence and to aid those in the firm when making more informed investment decisions regarding the re-architecture of the infrastructure system.

The results below are created using the Unique Services direct strategy while using the JRockit "GenCon" indirect strategy. The combination of the two produced the best results out of all strategies conceived which can be reviewed in section 8.6.

5.1 Insert

When an Oracle Coherence system first goes into production, data must be loaded into the cache from a persistent data source. In addition to standard JDBC load, Barclays also asked us to test their in-house tool for interacting with the database, called JEDI, to determine whether their investment in that particular project had been worthwhile.

The first result set gathered was the amount of time taken to load all caches with the appropriate amount of data found in a production environment. Initially, we did not use a database. We put the same object into the cache repeatedly, each time with a different key. This effectively isolated the time that is required by Coherence to load data into the cache, putting a lower bound on expected load time. We conducted this test for one million, three million, and five million rows of Position Data. The result can be seen below.

Affected Rows	Average Time (seconds)
1000000	22.518
3000000	76.064
5000000	141.210

Table 5.1: Loading One Cache From File

As you can see, the amount of time it takes to load a cache directly increases with the number of rows. This is what we would expect, and indicates that the data has ample resources for a cache of this size.

We also tested the amount of time it takes to load a cache from the database using two loading frameworks: JDBC, JEDI. As discussed in 4.4.2, JDBC and JEDI are two types of database frameworks integrated into our framework for loading purposes. To compare their effects we chose to pit these two loading schemes against each other. In addition, we compared the database loads to the file load to provide a comparable baseline.

Load Name	Average Time (seconds)
File load	22.518
JDBC	50.040
JEDI	54.194

Table 5.2: Load Latency Comparisons

The data in the table above serves to provide two separate conclusions. The first is the realization that one million rows takes approximately 50 seconds to load into the cache from an external database; a 122% increase over inserting all rows via a file. The second conclusion is specific to Barclays. Because JEDI was designed in-house and provides an abstract layer to database functions it creates an overhead. However, from this data, it becomes apparent that JEDI performs quite comparably to the primitive JDBC framework.

From the data above it was apparent that JDBC was most efficient when loading data from an external database. This conclusion opened up the following question: how long does it take to load one million rows with JDBC while the server is experiencing external traffic? The answer to that question was gathered by initiating our traffic generation software and loading one million rows. The total time taken to do so was 341 seconds; a 682% increase over a load without traffic.

5.2 Update

At the daily close of each worldwide market, any discrepancies between the persistent data source and the cache must be rectified through an update. It was important to the business to understand how much time updating would require, especially with background traffic running.

As trades are placed, data will go to both the cache and the database; however, initial plans indicate that updates during the day will only be against the database. At the end of the day, internal estimates expect that 40% of the data will need to be updated, while queries will still be running against the cache. (With three major markets, the cache will be active throughout the day and night.)

We tested two main strategies for updating: to create an entirely new cache and then delete the old one, or to update only the 40% of the data that changed. The results can be seen below.

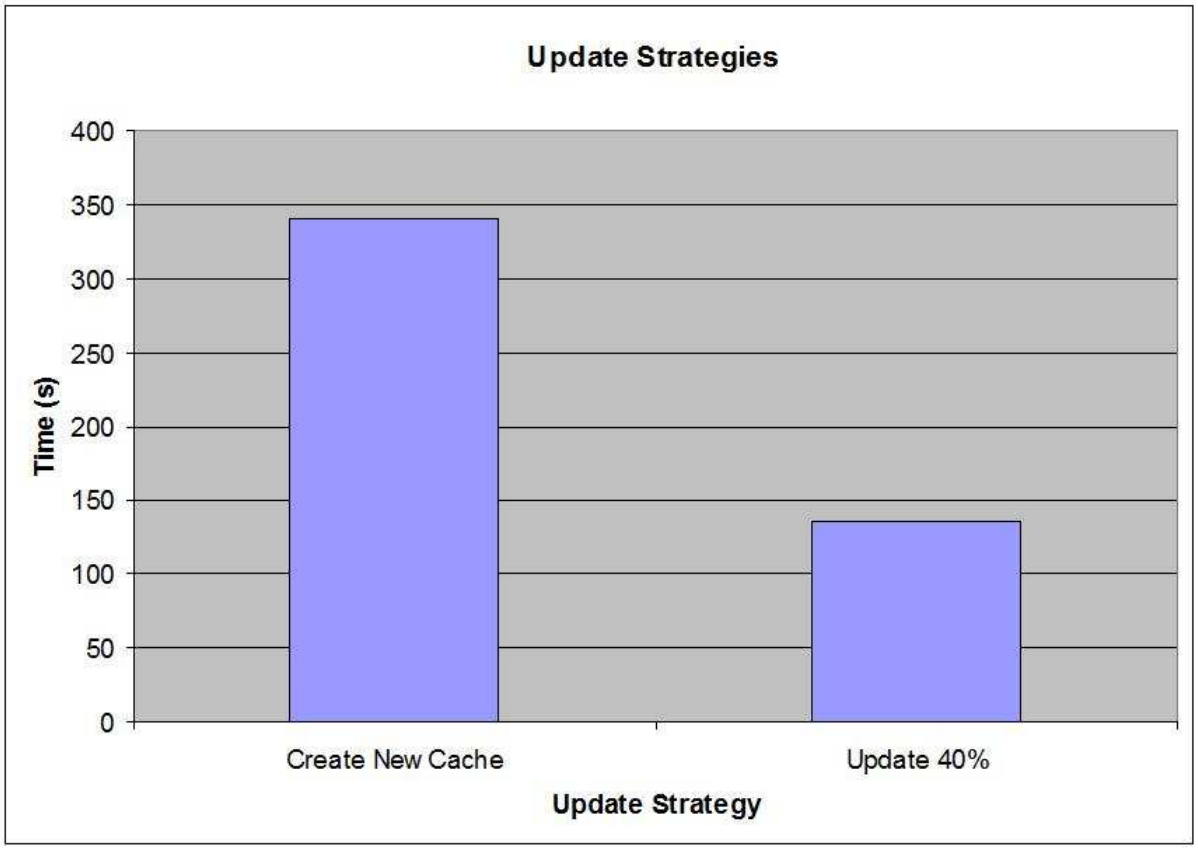

Figure 5.1: Update Strategies

As you can see above, it is significantly faster to update only the data that has changed, and this is the strategy that we recommend. Using the second strategy as opposed to the first with save computing resources when the close of market update is received each day.

5.3 Queries

The most common request performed against the cache will be queries made by downstream applications. Users of these applications such as Barclays' clients, will care about the response time of different queries. In this section, we captured the response time of a number of different queries to demonstrate the type of performance to expect.

The next test that we ran collected the baseline query times for each of the queries mentioned in section 4.4.1. Each query was first run in isolation, meaning the issued query is the only request that the Coherence cluster is servicing. Next, our traffic generation software was enabled and each query was again executed against the stressed system. The table below displays the results of our queries.

Query	Average Time without Traffic (seconds)	Average Time with Traffic (seconds)	Affected Rows
Position POF Key	0.516	1.560	914
Position Date	12.596	22.722	343909
Position Account(s)	0.406	1.444	914
Activity Key	.006	.007	1
Activity Settlement Date	0.471	1.575	147
Activity Trade Date	14.998	28.841	360671
Activity Trade Status	.397	1.807	1000
Activity Account(s)	.587	1.641	37
Activity Trade Status (3 Time Zones)	1.058	2.271	3000
Activity Settlement Date (3 Regions)	1.259	2.421	441
Activity Key (1000)	.021	.034	1000

Table 5.3: Baseline Query Times

The resulting comparison between queries with traffic and without is significant. All queries above show more than a two fold increase in average response time except for the ActivityKey query. A query against a unique key is extremely fast in Coherence (less than 1% of the time taken for other queries).

This query draws its benefits from the fact that Coherence's cache operates as a hash map. Thus, direct key accesses are much quicker and respond in a O(1) manner. While other searches become victim to linear search algorithms which operate in O(n). [1] As you can see at the bottom of the table above, even when querying 1000 primary keys, the performance is significantly better than other queries. For this reason, high performance systems using Coherence should utilize queries based on the primary key. These types of queries are not used in Barclays' current architecture, so we have outlined a possible architecture to use this in section 8.5.

5.4 Delete

To delete a cache, Oracle Coherence marks the memory used by the cache as free, and that memory is then reclaimed through garbage collection. This trivial operation of marking the memory as free completes in less than one second.

5.5 JVM Performance

Among the Coherence configurations tested we also chose to test the available JVMs and JVM configurations. The JVM has a large impact on operation of Coherence since it is responsible for such operations such as memory allocation, garbage collection, and many others. As discussed in section 4.7, through research we came across two possible JVM candidates: Sun Microsystem's JVM and Oracle's JRockit. In addition, we chose to run a strategy with JRocket's JVM argument "GenCon." The figure below demonstrates the response time of each JVM against six query types, three without traffic and three with.

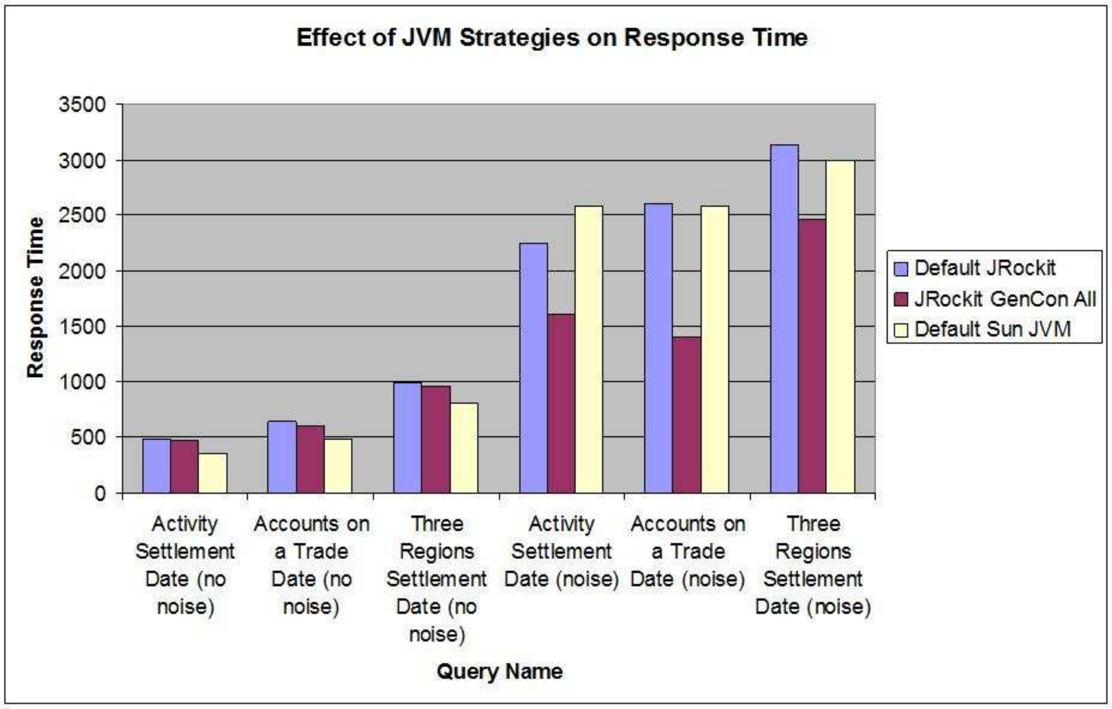

Figure 5.2: JVM Performance

The figure above demonstrates many interesting facts. The first fact is concluded from the three queries on the left of the figure above. These three queries are executed without external traffic stressing the cluster. The data shows that during these circumstances, the Sun Microsystem JVM consistently handles data much faster than the two other configurations. However, when we view the right side of the figure, queries with external system traffic, we notice a different story. This data set shows that Sun's JVM performs poorly when servicing multiple requests. Instead, Oracle's JRockit with "GenCon All" enabled consistently performed better than the others.

The figure above proves two things, but draws only one conclusion. Sun's JVM performs well in a no stress situation, however JRockit with "GenCon All" performs well under stress. However, a production system will operate with multiple users accessing multiple data sets. Thus, only the right hand side of the figure is valid for production, thus providing us with the conclusion that JRockit with "GenCon All" enabled will perform best in Barclays' new architecture.

5.6 Data Types and Performance

One of the most controversial topics during this project was assigning data structures within the Position and Activity objects the correct data type. Originally, many of the data types were classified as Strings because their database counterparts were also String based. While acceptable for storage means, it quickly becomes problematic when logical operations are performed on this data type. Because a String's complexity can very, comparing two requires byte by byte iteration over the two until a mismatch is discovered. The problem becomes more complex when a String is used in a greater-than or less-than operation with another String.

The solution to this problem is converting as many objects into number representations as possible. Integer operations are extremely fast and do not suffer from many of the drawbacks Strings have. In addition, Integers are found to be smaller in size than a String containing a written representation.

To prove this notion, we constructed a two queries. The first searched for a String object. The second searched for an Integer representation of the String in the former query. Both queries returned the same amount of rows. The table below depicts the result of the experiment.

Data Type	Duration of Search
String	14247ms
Integer	10906ms

Table 5.4: Query of a String vs Integer

The query against an Integer performed approximately 23.45% faster than its String counterpart. With this information, we urge Barclays to represent fields such as date, account Id, and other possible fields as Integers or Longs to reduce query time.

5.7 Fail-over Behavior

One of Coherence's selling points is that it preserves data under any single point of failure. This covers the failure of a single JVM, and also the failure of an entire server. We tested the affect fail-over would cause on the response time of the baseline queries and loading of one million rows. During out tests, we verified that none of the data was lost.

In one test, we were curious to see how the system would perform if we failed a server, let the cluster stabilize, and then failed another server. We began with eight servers, and did not experience significant problems until we had failed the fifth server. At this point, the size of all of the caches, including data replication, was 30 GB and there were only 36 GB of total heap space available across eighteen JVMs. As we saw, this was not enough available space; there was a series of Out of Memory Exceptions, and significant data loss. As a rule, we found that we should keep 30% - 50% of the heap space free in order to allow for a single server to fail and have the cluster still perform well.

To measure the effect on query response time, we picked a representative query and ran it repeatedly, measuring the response time as the server failed. We chose "Activity Trade Status" because it queries based on a String and returns a moderate amount of rows (1000). The results of a single test run can be seen below. It is important to notice the shape of the curve, including the variation in initial query times, the spike when the node is forcefully exited, and the oscillation in response time as data is repartitioned.

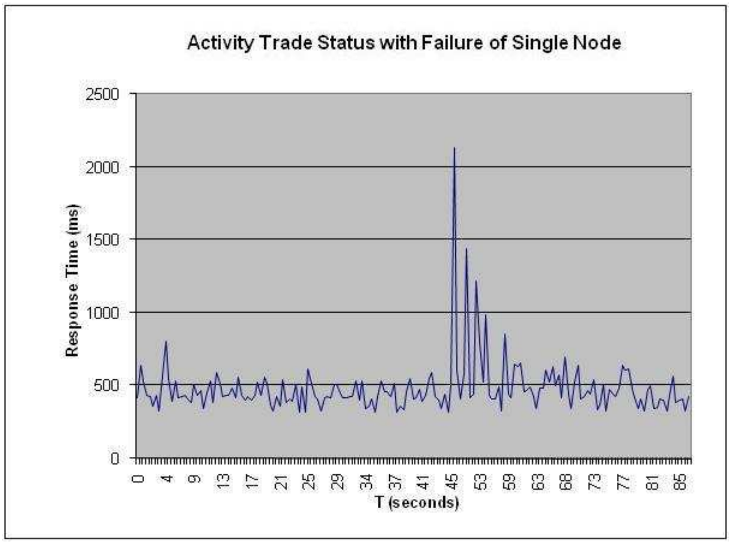

Figure 5.3: Fail-over of a Single Node (No Traffic)

At t=45 seconds, we forced one of the nodes to exit. As you can see, there is an initial spike in the response time of the queries to about four times the average. While the above results are for one run, repeat testing allowed us to conclude that this was typical. After about fifteen seconds, the data has redistributed and the response times return to normal.

We also ran the same test with a background traffic of ten users querying the cache continuously with a one second sleep between each query. The results of this test are shown below.

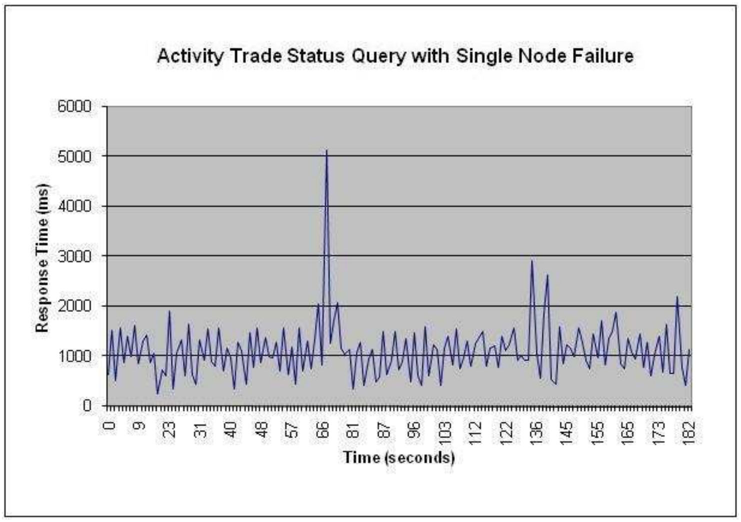

Figure 5.4: Fail-over of a Single Node with Traffic Generation

As you can see, there is significantly more variability in the response times when there is background traffic. The spike in response time is initially a five times increase in response time with a similar recovery time of about 15 seconds.

Similar tests were repeated when simulating the failure of an entire server. To do this, we forcefully exited all six of the nodes running on a single physical machine. In all cases, data was preserved. The results of this test run without external traffic is shown below.

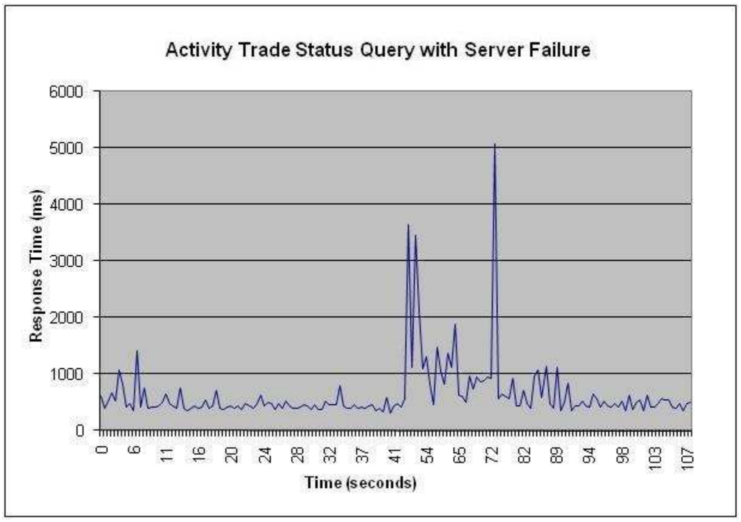

Figure 5.5: Fail-over of a Server (No Traffic)

At 42 seconds, we issued a command to forcefully exit all of the nodes on one of the physical machines. As you can see, this caused large spikes in response time. Recovery from the loss of six nodes at once takes the cluster about 40 seconds. The performance takes a more dramatic hit when there are queries running in the background. The results of this test are shown below.

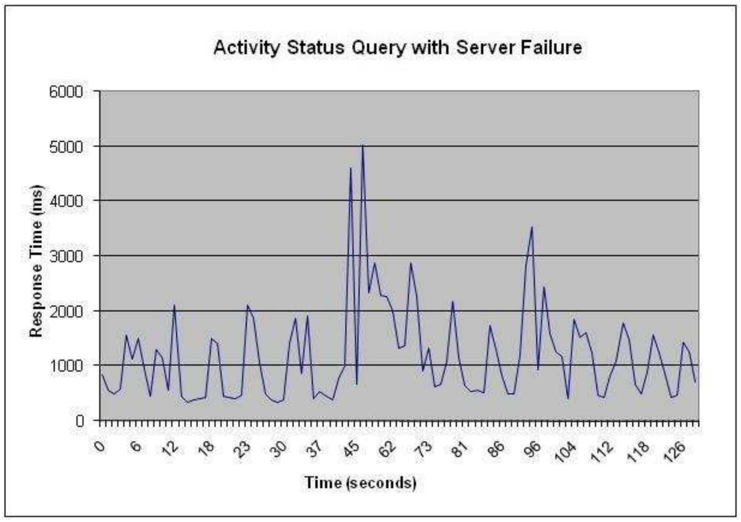

Figure 5.6: Fail-over of a Server with Traffic Generation

The effect of the traffic can be seen visually, where the response times of the query are large, there is more variation, and the effects of server failure continue for about 70 seconds.

The Coherence cluster is able to withstand failure of a single node or a single physical server without the loss of data. Given adequate time for data distribution, servers can continue to fail until the size of the cache, plus the replicated data, plus the overhead of running Coherence becomes greater than the amount of available heap space. As expected, the effects of many nodes failing at the same time is more pronounced than a single failure, and both types of failure is more pronounced when there are other requests on the cluster. This understanding will help Barclays understand the effects node and server failure will have on their production systems if a Coherence cluster is deployed.

5.8 WAN Testing

WAN testing generated a lot of interest within Barclays because it would be their first attempt to replicate cached data across the WAN. Sources within Barclays also stated that there is only one other instance of such a system in the financial industry.

The replication that we tested was a one way "push replication", where the cluster in New York City was configured to be the "active" cluster and the cluster in London was configured to be the "passive" cluster. In this setup, any modifications made to the cluster in NYC are propagated asynchronously across the WAN to the cluster in London. Coherence also supports two way replication, which is little more than two active-passive connections. [7]

5.8.1 Latency

As described in section 4.8 of our methodology, we tested the latency of the WAN with a specific data object designed for this task and a Passive Listener that attaches to the passive cache and allows us to manipulate any objects received. To determine the latency effects of the WAN, we ran the latency test when inserting data into the same cluster, when using push replication with another cluster in NYC, when using push replication with a cluster in Piscataway, NJ, and when using push replication with a cluster in London. The results of this are shown below.

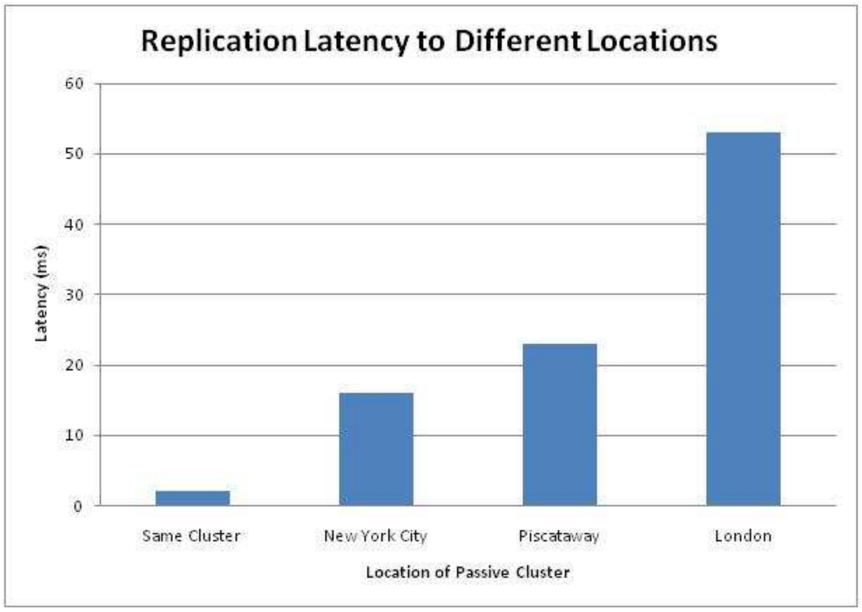

Figure 5.7: Latency Times

The first test, timing the insertion into the same cluster, was completed as a method of checking that our test worked correctly. This gave us a baseline number of 2ms, and is the average latency for a single object to be loaded into the cache within the same cluster (no replication). The next result shows the latency when replicating to another cluster in NYC. The difference between this result and the first one shows the overhead of push replication, because in both cases, the data begins and ends in the same data center in NYC. The third test, to Piscataway NJ, shows the effects of being routed through the WAN and to a different data center, albeit not far away from the originating data center. The final test, to London, demonstrates the effects of traveling across the Atlantic.

The latency between New York and London is the one that is most important because the goal is to replicate data between these two locations, however, gathering information about replicating to other locations helps to determine what components are responsible for the overall latency number.

We also tested the effects of object size on latency to London. The results can be seen below.

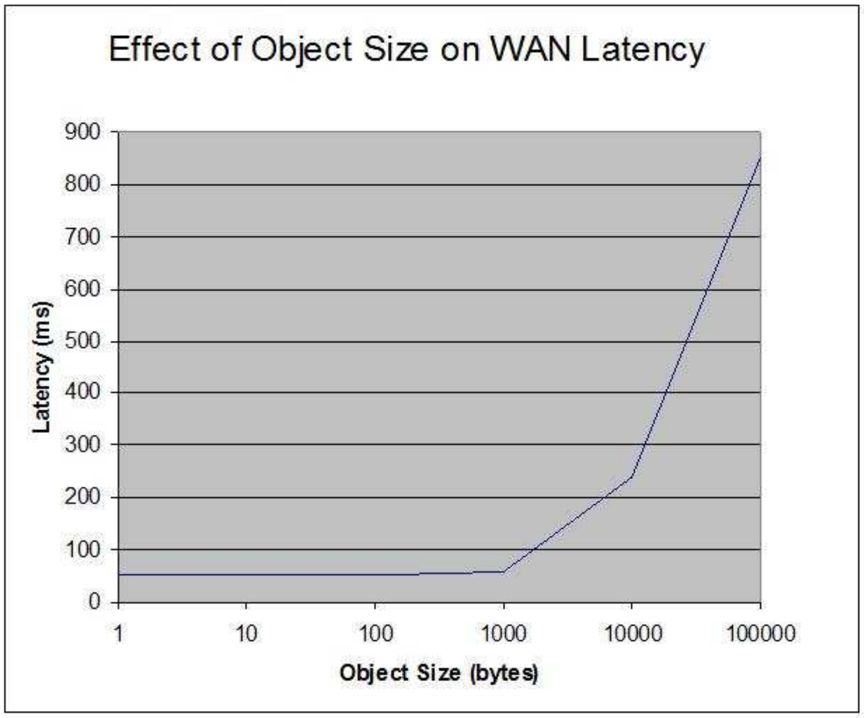

Figure 5.8: Effect of Object Size on WAN Latency

The objects that we tested for Barclays were less than 1kb each, so unless the size of the data objects is increased, Barclays should expect the latency to be 53ms.

One other result from our latency tests to note is that there were occasional spikes in latency time up to 150ms, although most times were closely distributed around 53ms.

5.8.2 Throughput

When using push replication, there are some configuration parameters that can be specified that affect the performance of the system. The two that we were primarily interested in were the delay and the batch size. This would allow us to chunk the data into batches and delay each batch in order to preserve WAN utilization.

Using the throughput testing strategy outline in section 4.8 of our methodology, we varied the batch size and delay parameters to gather the results consolidated in the following table.

	BatchSize	10	100	1000
Delay (ms)				
0		83	100	110
1000		77.6	83	73

Table 5.5: Effect of Batchsize and Delay on Latency

As you can see from the data above, we obtained the highest throughput with a batch size of 1000 and a delay of 0ms. From this data, we could conclude that more testing was required, especially with larger batch sizes and no delay, however, while doing this test we qualitatively observed that there was a significant impact on the NY load time and that we would need to devote our time to investigating this.

5.8.3 Effects on the Local Cache

When loading data into the cache with replication enabled, we noticed that load times were significantly longer. When we measured for comparison, we obtained the following results.

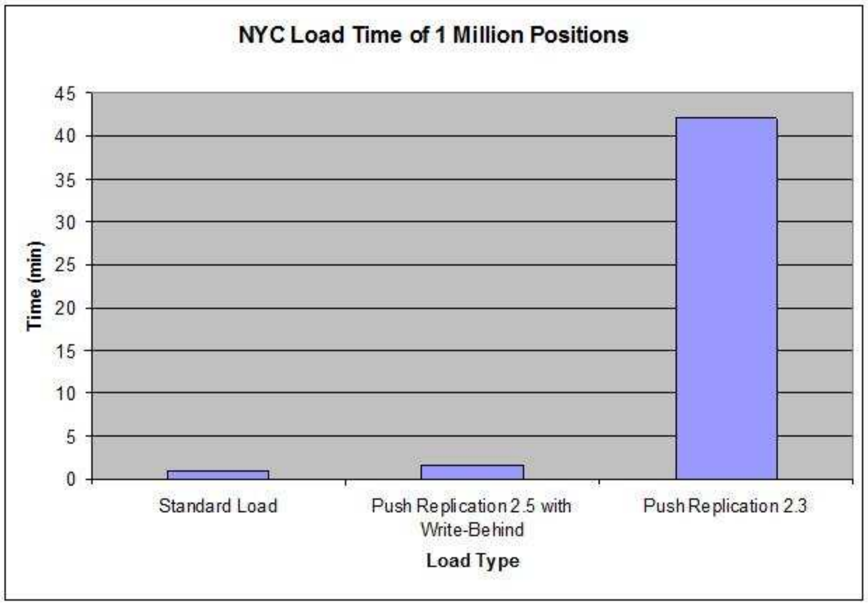

Figure 5.9: NYC Load Time of 1 Million Positions

The first test that we ran used push replication 2.3 and results in a forty-two times increase in the amount of time taken to load data into the NY cache. This is a significant penalty to pay for data replication that is asynchronous.

After reading Oracle documentation, we found that Oracle introduced a solution to this problem in push replication 2.4.0. [8] We upgraded our system to push replication 2.5, and noticed significant improvements. For comparison purposes, we have reproduced the graph above without push replication 2.3.

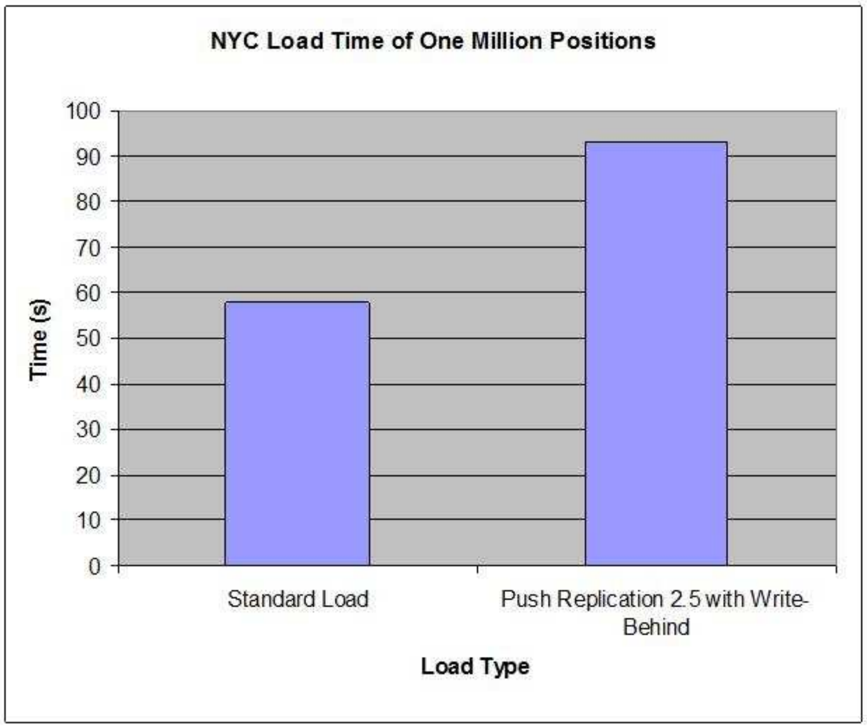

Figure 5.10: NYC Load Times of 1 Million Positions with Push Replication 2.4.0

The write behind feature of push replication 2.4 and 2.5 allows for configuration of a delay to be specified between the insertion of an object into the cache, and it being marked as being ready to be published. This extra stage drastically improved the performance of the active and passive clusters. The penalty paid locally (on the active cache) for push replication is now down to 60%.

This dramatic increase in performance with the upgrade from push replication 2.3 (currently the version utilized by Barclays) to push replication 2.5 led to another bottleneck in the system, an overloading of the socket connection on the passive site, leading to lost data.

5.8.4 The Use of Multiple Proxy Servers

After we realized that the lost data was due to an overflow of the socket buffer, we began to search for ways to remedy this. When Coherence uses push replication between two caches, a socket connection is open from the active cluster to a proxy server in the passive cluster. As we mentioned in section 3.5, we are using eighteen different caches. This means that eighteen socket connections are being opened, one for each cache. We realized that were are able to configure multiple proxy servers on the passive cluster.

We tested replicating all eighteen million data objects, one million in each cache. Sixteen million object were successfully replicated to London. We were able to determine that Coherence did not open an equal number of socket connections to each proxy server, overloading one of the proxy servers which caused our loss of data.

We created a workout for this that manually specified which proxy server each cache should connect to, and were able to replicate all 18 million rows between the NYC and London clusters. We recommend that the bank submit a bug report to Oracle. Oracle documentation states that the socket connections are made to a random proxy server, however, the desired behavior is to have the socket connections be evenly distributed amongst the proxy servers so that the load is balanced.

5.9 .NET Implementation

Coherence is native to the Java language, and is able to work directly with java clients. Coherence also allows clients to connect through a proxy server, which allows clients written in other languages, such as .NET, to communicate with the cluster.

The implementation of the .NET client provided interesting results when compared to its Java counterpart. The original assumption was that Java would respond faster due to the fact that Coherence is a Java native program. However, when we recorded the results the data pointed toward another conclusion.

Query	Java	.NET
Activity Key	3.7ms	3.5ms
Activity Accounts	525ms	530ms
Activity Settlement Date	480ms	500ms
Activity Trade Status	781ms	756ms

Table 5.6: .NET vs Java Client

As the table above demonstrates, the difference between the two languages is negligible. The .NET Coherence performs just as well as its Java counterpart. Thus, this data set proves there is no clear advantage that Java has over .NET when operating as a client. Because our original framework was written in Java it was concluded that its implementation would serve to provide results across Java and .NET implementations.

These results were important to Barclays because they are interested in using clients written in .NET, and questioned what type of performance to expect.

Chapter 6

Final Recommendations

From this project, we have gleaned substantial information about the way Oracle Coherence works and found a number of strategies to improve the performance of the standard setup of the Coherence Cache. Our tests were done in the type of environment that would be used in production. From our results, we have formulated a number of recommendations.

Our first recommendation is to implement an Oracle Coherence caching system to supplement the database. Response times from the cache are better than the database due to the fact that Coherence operates out of memory while the database must retrieve its data from the disk drive. In addition, we recommend using the Unique Services Coherence configuration with JRockit and the “GenCon” JVM arguments as proven by the results in section 5.

We also found significant differences in response time based on data types. Initially, all fields were designated as Strings. We suspected better performance from Integers and Longs, so we constructed a test and found that this provides a significant reduction in size and a 23% reduction in response time as demonstrated in section 5.6. We advise converting as many fields as possible from String to Integer or Long. For example, we recommend representing a Date field as a Long.

When analyzing the results of our queries, it became evident that queries based on a primary key can be fulfilled in a small fraction of the time taken for other more complex queries depicted by the table of queries in section 4.4.1. Unfortunately, a useful financial query is rarely done on a primary key. We advise analyzing the advantages and disadvantages of a system outlined in section 8.5, to determine whether or not Barclays can capitalize on Coherence's strength at querying for the primary key.

Our fourth recommendation is in regard to resource utilization. For the Coherence cache, we recommend keeping the cache size around 1/3rd of the total heap size available. This allows another 1/3rd of the available memory for the backup partitions, and 1/3rd allows for the overhead of running coherence and a margin of safety in case one of the servers in the cluster fails.

Finally, we recommend using a Limit Filter and paging to reduce the effects of garbage collection. A Limit Filter causes the cache to return the results of a query in groups. Without background traffic, query times were better without the Limit Filter, however, after introducing background traffic, the results favored using the Limit Filter. Since the system will typically be run with concurrent requests, we advise using a Limit Filter.

The above recommendations are practical recommendations based on the results we gathered while benchmarking Oracle Coherence for Barclays' specific needs. Our research indicates that implementing these recommendations will improve the results generated by the Oracle Coherence application. By hearing the recommendations issued above Barclays has since been able to identify Oracle Coherence as a key component in their network infrastructure which attempts to provide its users with a quicker means to retrieve data.

Chapter 7

Conclusions

At Barclays, there is an initiative to reorganize the way financial data is stored. They are moving from an ad-hoc system, where specific methods of storage were built for specific needs, to a consolidated system designed for wide usage throughout the company. In the new system, the company would like to enhance performance of information that is in high demanding by persisting it in a Coherence cache.

The primary objective of our project was to determine the optimal configuration for the Oracle Coherence cache. We performed an array of tests, and found ways to enhance system performance based on JVM usage, data type usage, resource allocation, among other strategies.

Our second objective was to determine the type performance to expect from the Coherence cache. We profiled a number of different common queries, loads, and updates both with and without background traffic. We are able to demonstrate the effects of background traffic, the effects of single node failure, and the effect of a server failure. We also verified that the Coherence cache performed better on the same queries than the database we were loading data from. This information can be used by the company when determining the value added by implementing a Oracle Coherence cache.

A third objective that was introduced early into the project was to provide a framework that can be extended to benchmark the performance of the cache against other data objects.

Two more objectives were introduced as stretch goals, to be completed only if we efficiently progressed through the above objectives. The first of these was to demonstrate that the cache can be integrated with .NET, and to show the quantitative benefits or using a near cache along with the partitioned cache. We were able to complete this.

The second stretch goal, and fifth objective overall, was to demonstrate that the system can self-replicate across the WAN with the UK and to benchmark the effect this has on performance. This was the first time this was attempted at Barclays, and if it is implemented in production, it would make Barclays only the second financial firm to implement this.

We have completed all of our base objectives and stretch objectives for this project. We have benefited from learning a new industry, a new technology, and working alongside technology professionals. In addition, it has since been acknowledged that Barclays is already benefiting from our results in this field. Several of the recommendations found in section 6 have already been introduced into production groups at Barclays.

Chapter 8

Appendix

8.1 Barclays

Barclays PLC, an international financial service provider, began trading in 1690. John Freeme and Thomas Gould originally started by trading as goldsmith bankers in London. About thirty years later, the banks' headquarters moved down the street, and began bearing the familiar symbol of the Black Spread Eagle on the front of the building. At the time, many people could not read, so buildings were identified by an emblem. A few years later, James Barclay, son-in-law of John Freeme, joined the firm. Despite the many acquisitions and mergers over the years, the name Barclay has remained.

Towards the end of the 19th century, the company had joined with 19 other privately owned banks to form Barclay & Company, and shortly after the turn of the century, the bank began trading on the London Stock Exchange. Through the first half of the 20th century, the bank continued expanding through the acquisition of other banks in the United Kingdom, along with banks in Africa and France. Barclays became the first bank to have a female branch manager and the first bank to have a computer center for banking. Barclays also launched the first UK credit card and a year later, in 1967, opened the first ATM machine.

Barclays continued to expand throughout the last half of the 20th century, and by the end, was the most valuable UK bank. It was also the first bank to have a website. Most recently, Barclay was in the news for its acquisition of Lehman Brothers, which it bought out of bankruptcy for $1.35 billion, only slightly more than the value of the New York real estate.

8.2 Distributed Systems

A distributed system is a collection of individual computers, each communicating with each other via a network medium. In a distributed system, a task is divided into multiple parts, each of which is delegated to a single computer to be solved. The solution is then consolidated among the affected computers and returned to the user.

A distributed system owes much of its strengths to its ability to scale horizontally. Scaling horizontally is the process by which adding more computers to a distributed cluster is favored over increasing the processing power of a individual system. The advantage to scaling horizontally is that each node, or computational system, may operate virtually independent without regard for its peer systems. Thus, individual systems may avoid input/output bottlenecks as well as resource thrashing; both of which are commonly found within centralized systems. [11] In addition, distributed systems tend to be fault-tolerant, a property which allows the entire cluster to continue operating as a whole despite a critical failure on an individual machine.

As an example, Folding@Home is a distributed computing project aimed at solving complex protein folding simulations. The Folding@Home project makes use of a distributed cluster constructed from a multitude of average personal computers. Once the application is downloaded on a client machine, the program proceeds to connect the machine to a cluster of distributed systems. The client then receives a folding task from the cluster to compute and report the result. The true power in distributed systems was seen on April 9, 2009 when Folding@Home reported the project had reached a combined 5.0 native PFLOPS never before achieved by a computing system. [2]

8.3 Grid Computing

Grid computing is a distributed system that shares data and resources across all peers. Computational grids are most commonly used when resources must be combined to handle the complexity, and size of an individual task. In this case, many grids are used in conjunction with an administrative system that will facilitate information passing seamlessly, efficiently, and transparently. Unlike the distributed computational systems mentioned previously, grid computing projects bring the power and resources of all the computers connected to the grid together as one, so that it acts as a unified system.

Grid computing is becoming increasingly important to perform cutting edge research, and is used in a multitude of applications including physics, biology, medicine, and computer simulation. The Large Hadron Collider at the European Organization for Nuclear Research (CERN) currently gathers annual data in the petabytes. It has a five tier system of computers on the grid, from the main computers at CERN that gather data to the fifth tier which is the individual researcher accessing the data grid from their personal computer. In biology, grid computing is being used to map genomes. Digital brain scans are being used, where a three dimensional brain scan requires a fraction of a terabyte of storage. Increasingly, systems are being investigated where patient data is shared over a grid, allowing for a more streamlined health care process.

There are many challenges that are faced when trying to construct a data grid. There must be appropriate distribution of resources, the grid must be able to adapt to have servers come online or go offline, there may need to be certain redundancy to prevent data loss and to help prevent bottlenecks where a server is waiting for data to travel over the network.

The nature of the grid may change over time, as hardware is upgraded, more servers are added, or server failure takes some of them offline. This requires that the data grid is adaptable to change. The data grid has great potential for solving complex problems and getting unprecedented performance, however due to the data grid's complexity, in order to maximize the results, the implementation of the grid is extremely important.

8.4 The Portable Object Format

Java provides its own method of serialization that can be used with Oracle Coherence, however, it has been found to be inefficient. Oracle introduced a language independent format called Portable Object Format, or POF. In our benchmarking suite, we made extensive use of POF. Based on our background reading, POF serialization uses significantly less time to transfer data (one-third to one-sixth the time). Based on this finding, we used only POF serialization.

POF has two primary speed advantages over the standard Java serialization. In this format, the storage size of data is reduced to its smallest form. For instance, if the number "3" is being transmitted as a long, it will not send the entire 64 bits, 62 of which are just zeroes. It will send only the necessary bits, and then a marker to show its data type. In some cases, such as this example, it can significantly reduce the size of the object, and therefore, the transmission time of the data over a network.

The second way that POF allows for improved performance is through quick extraction of certain fields. When defining a POF, the code explicitly states what attributes the object has, and the order that they will be serialized and de-serialized in (the order must be the same for this to work). When searching for specific data, you can use something called a PofExtractor, which takes the data type that you are extracting, and the location (index) in the POF that the specified field is stored in. For example, if you are querying Activities for a specific "account id", then you would create a Pof Extractor that would look for a string that matched the account id you are looking for. You would give this PofExtractor the location where "account id" is stored inside of the ActivityTypePof. This would allow the extractor to look only through account ids, which in our case, is a tremendous savings in the amount of data to be processed. [10]

8.5 Capitalizing on the Primary Key

In the "Results" section, we learned that there are significant advantages to querying the cache based on a primary key, often times requiring less than 0.5% of the time of other queries. Unfortunately, the design of the current system will be unable to take advantage of this tremendous performance. We recommend profiling the types of requests that currently get handled by the production database and designing a system to take advantage of the primary key.

Profiling and redesigning the system was beyond our capabilities given the time constraints of our project, however, we came up with some ideas while we went about our work that may be of use. This design is based on the following assumptions, which may or may not be correct and would depend upon the specific application.

- Eighty percent of the queries run against only ten columns in the database.
- Downstream application is only interested in entire object 5% of the time.
- Downstream applications display 100 objects at a time 90% of the time.

8.5.1 Design

Create a smaller version of the object with only 10 attributes (a 90% reduction in size). Have two caches, one with the entire object and one with small caches. Have control logic in front of the cache to determine whether to query the large or small objects.

8.5.2 Further Assumptions

Again, these may or may not be true. They are estimated based on the results we have seen, however, they were not deduced directly from data. These assumptions are simplifications for demonstration purpose.

- Assume the average query takes 10 seconds.
- Assume limiting a request to 100 objects provides a 90% reduction in time.
- Assume reducing the size of the object provides an 80% reduction in response time.
- Assume a query on a primary key takes 1ms for all objects.

8.5.3 Benefits

We will calculate the new weighted average response time based on the stated assumptions.

Contribution of 20% of queries that go against the larger objects: RT = (10s)(.2) = 2s

Contribution of queries that are not limited. This query should go against the larger cache. RT = (10s)(.8)(.1) = 0.8s

Contribution of queries that are limited to 100 and the application is only interested in the smaller object: RT = (10s)(.8)(.1)(.95) = 0.76s

Contribution of queries that are limited to 100, but require the large object. We pay a 1ms penalty for the primary key retrieval of each larger object. RT = ((10s)(.8)(.1) + .1)(.05) = 0.045 s

Total time: RT = 2s + 0.8s + 0.76s + 0.045s = 3.61s

This is almost a 64% reduction in average response time, given the above stated assumptions.

8.5.4 Disadvantages

With this extra reduced size cache, there would be an increase in required memory by about 10%. Also, updates made to one cache would need to be replicated to the other cache.

8.5.5 Summary

This system takes advantage of the Oracle Coherence's high speed query on primary key, the dependency on network latency and returned query size, and the usage a smaller, secondary object. We do not know what the query profile for Barclays is, and we do not have the capability to find it out. We do recognize, however, that there could be opportunities to take advantages of faster caching systems, and this example may be one such system.

8.6 Strategy Comparison

Please see spreadsheet documents submitted with the project.

Bibliography

[1] Algorithms: Big-oh notation. Available from: http://leepoint.net/notes-java/algorithms/big-oh/bigoh.html.

[2] Folding@home. Available from: http://folding.stanford.edu/English/FAQ.

[3] Robert Englander. Developing java beans, 1997. Available from: http://oreilly.com/catalog/javabeans/chapter/ch01.html.

[4] Apache Software Foundation. Jmeter. Available from: http://jakarta.apache.org/jmeter/.

[5] Sun Microsystems. Java hotspot at a glance. Available from: http://java.sun.com/javase/technologies/hotspot/.

[6] Rob Misek. Types of caches in coherence, June 2009. Available from: http://coherence.oracle.com/display/COH35UG/Types+of+Caches+in+Coherence.

[7] Oracle. Push replication pattern. Available from: http://coherence.oracle.com/display/INCUBATOR/Push+Replication+Pattern.

[8] Oracle. Push replication pattern 2.4.0. Available from: http://coherence.oracle.com/display/INCUBATOR/Push+Replication+Pattern+2.4.0.

[9] Oracle. What oracle coherence can do for you. Available from: http://www.oracle.com/technology/products/coherence/coherencedatagrid/coherence_solutions.html.

[10] Oracle. Pofextractors and pofupdaters, October 2009. Available from: `http://coherence.oracle.com/display/COH35UG/PofExtractors+and+PofUpdaters`.

[11] Hagit Attiya Jennifer Welch. *Distributed Computing*. Wiley, 2004.