Automated Application Profiling and Cache-Aware Load Distribution in Multi-Tier Architectures

Rizwan Maredia

School of Computer Science

McGill University, Montréal

August 2011

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements of the degree of

Master of Science in Computer Science

Abstract

Current business applications use a multi-tier architecture where business processing is done in a cluster of application servers, all querying a single shared database server making it a performance bottleneck. A prevalent solution to reduce the load on the database is to cache database results in the application servers as business entities. Since each of the in-memory application cache is small and independent of each other, a naïve load balancing algorithm like round-robin would result in cache redundancy and lead to cache evictions. By clustering these caches, we get a distributed cache with a larger aggregate capacity, where an object is retrieved from the remote cache if it is not found in local cache. This approach eliminates redundancy and reduces load on the database by a great extent. However, accessing remote objects incurs network latency affecting response time.

In this thesis, we transform the distributed cache into a hybrid one that supports replication so that popular requests could be served locally by multiple application servers. We take advantage of this hybrid cache by developing a holistic caching infrastructure. This infrastructure is comprised of an application monitoring tool and an analysis framework that work continuously alongside live application to generate content-aware request distribution and caching policies. The policies are generated by request-centric strategies that aim to localize popular requests to specific servers in order to reduce remote calls. These strategies are flexible and can be adapted easily for various workloads and application needs. Experimental results show that we indeed derive substantial gain in performance using our infrastructure. Our strategies resulted in faster response time under normal workload and scaled much better with higher throughput than existing approaches under peak workload.

Résumé

Les applications commerciales courantes utilisent une architecture multi-tiers où le traitement logique est effectué en un groupe de serveurs qui accèdent à une seule base de données partagée, ce qui la rend un point d'encombrement. Une solution répandue qui réduit la charge sur la base de données est la sauvegarde des résultats de requetes à la base de données au niveau des serveurs d'applications comme des entitiés logiques. Tandis que chaque cache local de chaque serveur est limité et est indépendant des autres, un algorithme naïve de balancement de la charge, comme round-robin, résultera en des duplications de copies dans les différents caches et mènera à des explusions de ceux-ci. En regroupant ces caches, nous formons un seul cache distribué avec une large capacité, où un objet est extrait à partir d'un cache distant s'il n'est pas trouvé localement. Cet approche élimine la redondance et réduit considérablement la charge sur la base de données. Cependant, accéder à des objets distants encours une latence au niveau du réseau ce qui affecte les temps de réponses.

Dans cette thèse, nous transformons le cache distribué en un cache hybride qui supporte la duplication ce qui permet de servir les requêttes les plus populaires localement par plusieurs serveurs d'applications. Nous prenons avantage de cette structure hybride du cache en developpant une infrastructure holistique du cache. Cette infrastrcuture comprend un outil de surveillance et une infrastructure d'analyse qui fonctionne d'une façon continue et parallèle avec l'application afin de générer un contenu qui prend en considération la distribution de requêtes et les politiques du cache. Les politiques sont générées par des stratégies orientées requêtes qui visent à localizer les requêtes populaires à des serveurs spécifiques et ce pour réduire les appels distants. Ces stratégies sont flexibles et peuvent être ajustées facilement pour

different charges de travail et besoins d'applications. Des résultats expérimentaux montrent qu'effectivement nous dérivons un gain substantial en utilisant notre infrastructure. Nos stratégies ont resulté en des temps de réponses rapides sous une charge de travail normale et donnent des bons résultats lors d'un débit élevé comparativemnt à d'autres approches sous des charges de travail de pointe.

Acknowledgements

I am exceedingly grateful to my supervisor Prof. Bettina Kemme for her continuous guidance and financial support. Her deep knowledge of the domain has always been inspirational while her insightful ideas have kept me motivated throughout my research.

I was very fortunate to have the support of talented researchers around me. I am especially thankful to Kamal Zellag, Sanket Joshipura and M. Yousuf Ahmed for our thoughtful discussions and collaborations. I would like to thank Andrew Bogecho and Kailesh Mussai for their technical support.

And last but not the least I am thankful to my parents and siblings for their encouragement, affection, and moral support.

Table of Contents

List of Figures

Chapter 1

Introduction

Internet applications have seen a tremendous growth in the last decade, and have become overly complex. E-commerce applications such Amazon and EBay have all faced common challenges like performance, reliability and scalability as the user base grew. To overcome some of these challenges internet applications are be broken down into multiple tiers, mainly consisting of the client tier, the application tier and the data tier. Each tier interacts with the next tier in line and sends result back to its previous tier, which makes application more flexible and manageable. However, a multi-tier architecture is not always robust and scalable to an unpredictable workload. Some of the concerns of today's internet applications are to be up and running 24/7 serving millions of customers and managing terabytes of data. For example, CNN on Election Day had a record 276 million page views and 27 million unique visitors [1]. Thus, it is clear that manual optimization of internet application is not easy.

To cater to increasing web traffic administrators can either update existing hardware components or add another server to share load. The later approach, commonly known as clustering, is becoming a norm, where a load balancer distributes the load (requests) to multiple servers. Usually, the application tier is clustered using

commodity machines while the data tier is not as data consistency is more complex. The data driven applications tend to query the database more frequently. This creates a bottleneck at the database server, and adversely impacts application throughput. Increasing cluster size cannot solve this problem because application scalability is curtailed by an overloaded database.

To reduce the load on the database server, the application server caches database results in memory so that subsequent requests for the same data can be served quickly from the cache saving trips to the database. This works out for smaller and consistent workloads but if the workload is big and is accessing a large data set the cache fills up quickly. This problem is more imminent when the load balancer sends requests to the servers in a round-robin way causing each application server cache to have nearly the same popular objects. Once the cache is full, requests which cannot be served from the cache, have to go to the database, and then are stored in cache causing evictions of previously stored objects. This poor utilization of valuable memory for a standalone cache got the research community thinking and led to the emergence of a distributed cache, where objects are transparently stored and retrieved from local or remote caches. Hence, the aggregate capacity of a distributed cache grows as nodes are added to the cluster. This solution also eliminates redundancy in caches.

Distributed caching allows applications to scale easily but does not always result in optimal performance. This is mainly due to the latency involved in retrieving objects from remote cache. In this thesis, we propose a solution that can reduce remote calls greatly. Adhering to the principles of *separation of concerns* our solution is designed to be highly decoupled and independent of the particular web application. We have developed a caching infrastructure that transparently monitors the application, performs periodic analysis and generates content-aware load distribution and caching policies with the intent to maximize the local cache hit-rate. When the load balancer is

equipped with our rule-based distribution algorithm, it chooses the server that can serve the request mostly if not entirely from the local cache.

Another important concern that we deal with in this thesis is distribution of popular content. News and e-commerce websites often receive more requests for some content than others. Thus, we prioritize requests on their access frequency and generate policies for the most frequent ones. Assigning a popular request to a specific server can create imbalance in load distribution where a server that caters to popular request gets saturated while others are underutilized. We alleviate this imbalance by adding replication support to our caching infrastructure so that popular objects are accessible from multiple servers. The top most popular requests can now be assigned to multiple servers for better load distribution.

1.1 Contribution

The noteworthy contributions of this thesis are:

- We design and implement a flexible infrastructure that transparently logs application and cache accesses to a remote server for processing and analysis.

- Leveraging the analysis framework, we design and implement of a set of request-centric strategies that find popular requests and assigns them to specific servers to maximize local cache hit-rate.

- We provide an extensive evaluation of our strategies and comparison with other caching schemes.

1.2 Thesis Outline

The rest of the thesis is organized into the following chapters:

- In Chapter 2, we provide the background information related to our research, specifically on multi-tier architectures and caching. We also discuss some related work that has been an inspiration towards this research.
- Chapter 3 deals with the infrastructure of our monitoring and analysis framework.
- In Chapter 4, we present the essential strategies that consume processed logs to produce content-aware policies for the application.
- In Chapter 5, we demonstrate the effectiveness of our infrastructure and strategies using a popular benchmark.
- We finally draw conclusions in Chapter 6 and propose future enhancements to the framework.

Chapter 2

Background and Related Work

2.1 Introduction

In this chapter we give an overview of related technologies, architectures and frameworks upon which our thesis is based on. We start by explaining the architecture of web applications, and then we explore how to scale and improve performance of these applications using clustering, load balancing and caching. Lastly, we discuss distributed caching and other related work that are closely related to our research.

2.2 Web Application Architectures

A web application (aka Internet Application) is an application that runs on a web server and is accessed through a web browser (such as Internet Explorer or Firefox) over the Internet. In contrast, a desktop application is a self-contained program that runs in one's own computer and does not require Internet connection. The key difference here is the machine where the core business logic is executed. Desktop applications have to be installed on local hard drives before they can be run where as web applications are merely accessed by typing the application URL (like www.example.com/do/this/). Figure 2.1 illustrates this difference.

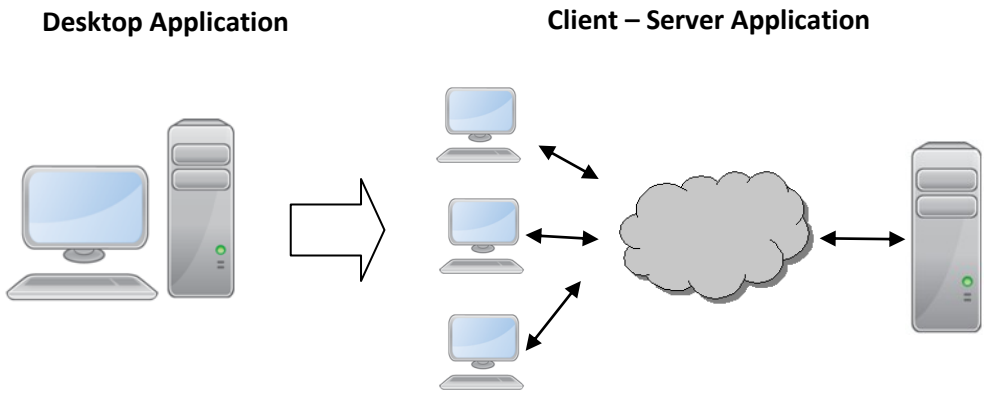


Figure 2.1: Shift from desktop to client-server application

During the 1990s most of the enterprise applications were built as desktop applications but since 2000 companies are moving towards web applications even for private applications because of the following advantages.

- Web applications enable multi-tenancy (i.e. a single application is accessed by users all over the world).
- The user requires only a thin client like a browser to access a web application; so even an outdated machine or a low-end device can access them.
- Any upgrades to web applications are transparent to the end user. The user simply sees new functionality as it becomes available.

These advantages come with certain challenges for the application provider. Architecting a web application is not as easy as a desktop application because it follows the request-response model as opposed to an event-based model of a desktop application. When a user enters a URI the request goes to the web server, where it is processed, and results are shown back to the user as response. This approach, which is commonly known as the client/server model, starts as a 2-tier architecture but can extend up to multiple tiers.

2.2.1 3-Tier Architecture

To enable multi-user tenancy on the Internet, web applications need to recognise each user and provide services based on his preferences, personal information and the request itself. For this, a web server maintains a web session for each user keyed through a unique session id. This session id is then passed along in each subsequent request/response pair for identification. The session itself may store data such as shopping cart items. Moreover, applications also need to store and retrieve data from a data store such as a relational database. Just the way a client makes calls to a web server, the web server makes calls to the database. Our simple 2-tier now becomes a widely popular 3-tier architecture illustrated in Figure 2.2.

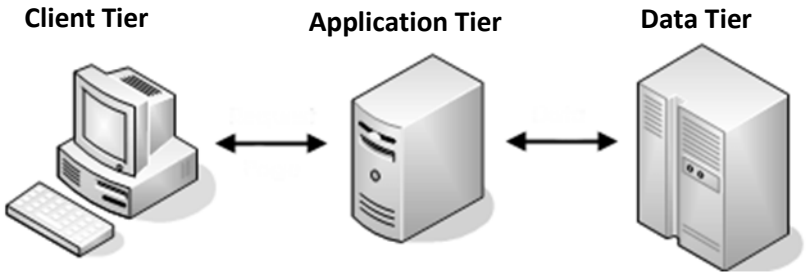


Figure 2.2: 3-Tier architecture

The client tier remains intact, but we split the server tier into an application tier and data tier. These tiers can be described as follows

- **Client Tier:** The client tier comprises all remote clients having access to the web tier over the internet. This is not just limited to web browsers, but any application, deployed in any form factor such as PDA or smart phones that is able to send requests to the web tier using the HTTP protocol is considered a client or more specifically, a web client. In today's service oriented web environment, Internet protocols have evolved and are not restricted to basic

HTTP for communication. Protocols such as SOAP (Simple Object Access Protocol), REST (Representational State Transfer), RSS (Really Simple Syndication) and RPC (Remote Procedural Calls) have extended the capabilities of web communication, and thus, web clients have become richer in the way they consume and represent information to the user.

- **Application Tier:** The application tier (aka business tier) processes the client's request and then responds to it over the same communication channel established by the client. Conceptually, it contains a web server that processes http requests. To the application tier, each request is considered unique. Thus, as discussed earlier, modern web servers provides cookie and session management features to recognize clients. This tier executes business logic on the request and composes a response (sometimes specifically tailored for each client's display capabilities and protocol involved). Request to static resources such as images and binary files are served without any business processing. Requests that do require processing are considered *dynamic requests.* Usually, the web application contacts the data tier to retrieve request pertinent information.

- **Data Tier:** The web application stores user specific or business specific data in the data tier. Most commonly a relational database management system (RDBMS) is used to store and retrieve information from the database – the persistent storage component of a data tier. XML, native files, persistent maps and registries could also be used for the data tier. The data tier also exposes high level APIs (or low level driver interfaces) to the web application. For example, ODBC (Open database connectivity) and JDBC (Java database connectivity) are APIs to communicate with the relational database using the SQL query language.

2.2.2 N-Tier Architecture

The 3-tier architecture presented above was fundamentally constructed to reduce the complexity of *Business-to-Consumer* (aka B2C) applications. Here, the consumer is the client and the business corresponds to the web application and its data. Soon the need for a *Business-to-Business* (B2B) application appeared, involving service level agreements (SLA) and demanding complex interactions between businesses, not sufficiently addressable by a B2C system. These requirements and more paved the way for web-services which eventually evolved into a Service Oriented Architecture (SOA). In SOA, requests from clients (users or applications) are directed to web-services that will verify the client's credentials before accessing enterprise data. The B2B systems that catered to other businesses as well as general consumer popularly became known as Enterprise Information Systems.

Technically, the architecture of an Enterprise Information System is very complex and varies for each enterprise, but conceptually it simply extends the 3-tier architecture into n-tier architecture by:

- Separating the application tier into web tier and business tier. The web tier interacts with the client, maintains user's sessions, and dynamically generates content from the results processed by the business tier. The business tier comprises several business components to do business processing on given inputs. It is oblivious of the web environment and therefore, provides flexibility for other systems to connect to it.

- Incorporating legacy systems, databases, ERP systems into an Enterprise Information tier. These data sources are accessed by specialized components in the business tier.

Several venders now offer enterprise frameworks that are typically modelled after such n-tier architectures. One of the popular frameworks is Java Enterprise Edition (JEE) [2], whose specification provides APIs (and reference implementations) of various components to rapidly build enterprise applications. Figure 2.3 illustrates the Java EE n-tier architecture which adheres to this approach.

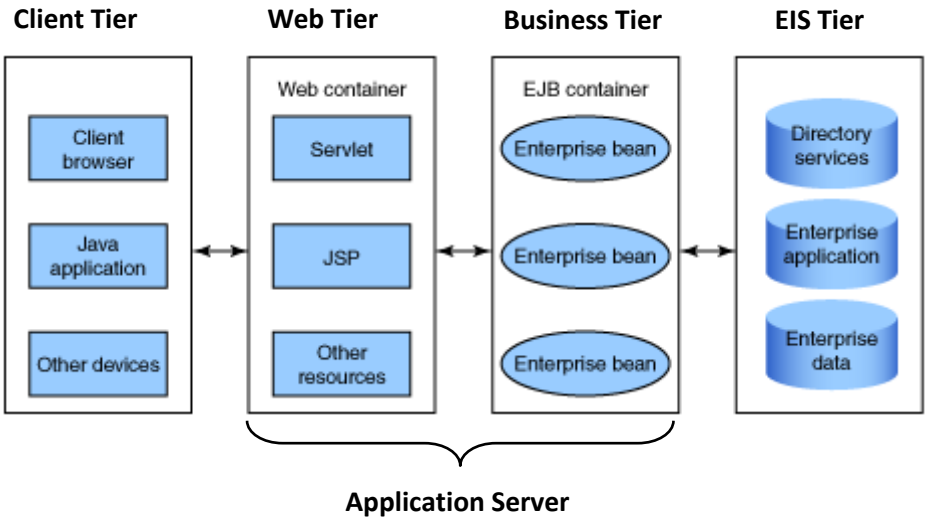


Figure 2.3: Java EE n-tier architecture

The web tier and the business tier form the application server and are typically run on the same Java virtual machine (JVM). In Java EE parlance, an application server provides managed environments for different components through specific containers. The container is responsible for a component's lifecycle (from instantiation to destruction), request delegation, cross component interaction, and other things. For example, the web container is the interface between web components (Servlets, JSPs) and the web server. An EJB container manages enterprise java beans (EJB) [3], which provide the business logic. The EJB container makes it easier for enterprise

beans to handle cross-cutting concerns such as persistence, transactional integrity, and security – the discussion of which is beyond the scope of this thesis.

Since web tier and business tier are part of the application server some architects consider this separation as a *Multi Layer Architecture*. In this approach they view an application as separate layers of functionality performing a dedicated task in request processing. The typical layers in an application are:

- **Presentation Layer:** This layer simply corresponds to the web tier in our n-tier architecture, and is responsible for content generation.

- **Business Logic Layer (BLL):** It contains an application's core business logic, and it exposes services for B2B interactions. It includes Session EJBs which encapsulates business logic, web services for interfacing, and other utility modules.

- **Domain Layer:** It represents the business data model and therefore it is sometimes called the model layer. For example, a customer, account, and customer-account objects are part of the domain model of a financial application. The business logic layer performs its operations on the data in the domain layer.

- **Data Access Layer:** This layer provides connectivity to data residing in the Enterprise Information System (EIS Tier). It also conceals boilerplate code such as connection management, transaction management, and caching from the business layer. Hibernate, TopLink, and Java Persistence API (JPA) are examples of the data access layer. These frameworks provide a high level view of business data and are described in the next section.

2.3 Object Relation Mapping

As we have discussed before, the database is an integral part of a data driven application. The database provides mechanisms to create, read, update and delete (CRUD) data which is stored in a database table. A table is a collection of records where each record is represented by a row in the table. Each row has a fixed number of attributes addressable by their column names. A record in a table is identified by a unique key, called *primary key*. Relational databases go a step forward and allow database designers to specify relationship between data elements using key constraints. A primary key in one table becomes the foreign key in another table to establish a relationship between two tables. For example, the tables in Figure 2.4 illustrate the relationship between Categories and Items in an e-commerce store.

Categories

id	name
1	Sports
2	Electronics

Items

id	category_id	name
1	**1**	Football
2	**1**	Basketball
3	**2**	IPod

Figure 2.4: Database tables showing primary-foreign key relationship

The Category *id* appears in the Items table as foreign key *category_id*. It essentially captures a business relation that an item has an associated category. The application can now retrieve items in a given category using an SQL query like

***select** * **from** Items **where** category_id = 1*

The database vendor provides programmers with an API to perform CRUD operations using SQL statements. The business logic then has to extract each row from the result set, map each column into an appropriate data type, perform the operation, and perhaps execute several other queries to complete a business transaction. This leads to bloated application code, where simple business logic gets convoluted in chunks of database operations. More importantly, the application is not able to visualize the business operation as an interaction between business entities.

As most of the web applications are designed and implemented using object oriented languages (and architectures), there seemed great disparity between the relational view of data and the object-oriented view of business entities. To overcome this problem, *Object Relational Mapping* was introduced, which essentially translates (maps) relational data into real world business entities. Thus, an application can now concentrate more on domain models and their interaction using object-oriented language features. The domain layer now reflects database tables, and business layer can easily manipulate models without caring for intricate low level details.

The ORM transformations are done in the following way:

- For each table in the database an associated *Entity Class* is created.

- The columns in the table becomes the instance variable of the entity class. These variables are generally kept private and are accessed/modified using GET/SET methods. The data types of variables are language specific native or object types (Integer, Double, String, Date), and are type compatible with database data types. Usually, the length constraints of VARCHAR data types are ignored and enforced by validation rules.

- The primary key variable is explicitly identified. The foreign key variable is actually the reference of the entity class in case of one-to-one relationship, or a set of entity classes in case of one-to-many relationship. The inverse relationship is also created.

The UML class diagram in Figure 2.5 depicts how the Categories and Item tables in Figure 2.4 are mapped to business entities.

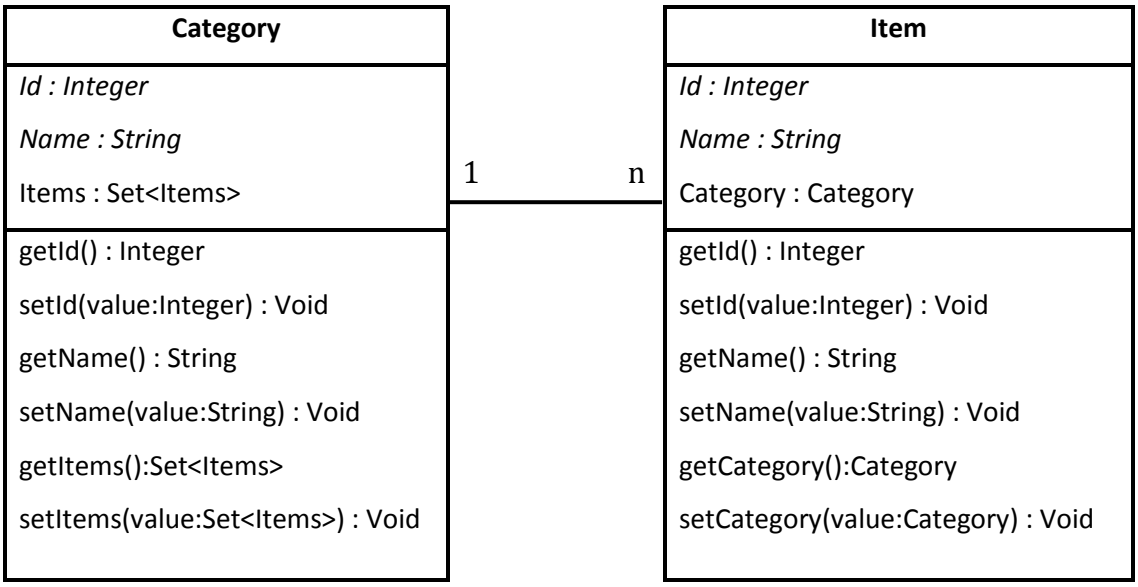


Figure 2.5: Entities class diagram

As you can see the Category class now has a reference to Items even though the Categories table does not. This is a really powerful feature which explicitly creates strong relationships, making it easier to access associated entities and children. Now we can access all items for a particular category using simple method calls:

```
Category category = Session.load(Category.class, 1);
Set<Items> items  = category.getItems();
```

The ORM framework does the heavy lifting of generating SQL queries when needed. Each ORM framework comes with a set of tools that allows programmers to generate Entity Classes from database tables. It also generates associations based on SQL schemas. The metadata for the object-relation mapping can be specified in XML files or embedded right into Java classes using Java Annotations.

There are two widely used ORM frameworks in Java EE, namely the Java Persistence API (JPA) [4] and Hibernate [5]. JPA itself is a persistence framework that specifies how entity classes are stored and retrieved from the database. JPA 1.0 offers the Java Persistence Query Language (JPQL), which resembles SQL but operates on entities rather than database tables. JPA 2.0 also supports type safe Criteria Query which is an object oriented query language that avoids incorrect query construction. JPA comes with a reference implementation of TopLink Essentials. Hibernate, on the other hand, has its own framework and query language - Hibernate Query Language (HQL) and Criteria Query. As of version 3.2 it also provides an implementation for JPA.

ORM frameworks also optimize database accesses. For example, when reading an entity Hibernate avoids fetching its associated entities unless explicitly accessed by the program. Hibernate also keeps track of changes made to an entity so that only modified entities are persisted to the database. Many ORM frameworks maintain a pool of active database connections to avoid connection setup latency for each database access. Moreover, these frameworks provide extensive support for entity caching which is discussed in Section 2.5.

The downsides of ORM frameworks are: slightly slower performance due to query translation and a little bit more memory requirement. Nonetheless, the advantages and optimizations provided by these frameworks far outweigh minor drawbacks.

2.4 Performance and Scalability

Performance of a system is a subjective matter and cannot simply be put in numbers without taking into account user expectations. A system that is responding unnoticeably faster compared to the one responding slightly slower may be considered performing equally well by the user. Thus, a system that meets user expectation is deemed well performing. System performance plays a key role in terms of web applications. There are several interrelated metrics that can define application performance such as:

- **Response Time:** How quickly an application responds to requests. Request time could be measured at each component level or from the beginning of the request to the completion of the response. Generally the faster the better.

- **Latency:** Latency affects the response time and is considered an undesirable feature of any hardware or software component. Some hardware components, such as network switches, have a predetermined latency which cannot be avoided. Software components however, can be optimized to minimize latency.

- **Throughput:** Throughput denotes the number of successful events achieved per unit of time. For example, the throughput of a network switch is the number of packets successfully routed per second, and the throughput of a web application is the number of requests successfully served per second. Throughput is often measured under peak load to achieve an upper bound value, what is called Maximum Throughput.

The ideal system would like to maximize throughput, and minimize latency and response time. Under stressful conditions (peak load), throughput could degrade and affect response time as component latency is increased. Usually throughput is limited by what we call the *Bottleneck*. Wikipedia defines bottleneck as

> "**Bottleneck** is a phenomenon by which the performance or capacity of an entire system is severely limited by a single component. Formally, a bottleneck lies on a system's critical path and provides the lowest throughput. As such, system designers will try to avoid bottlenecks and direct effort towards locating and tuning existing bottlenecks. Some examples of possible engineering bottlenecks are: processor, a communication link, a data processing software, etc"

Once a bottleneck is reached the system reaches a saturation point where performance can no longer be improved with the existing software and hardware configuration. With the widespread use of the Internet in the 21st century, web applications have encountered serious problems with regard to ever increasing web traffic. The applications that were designed to serve 500 requests per seconds started to choke or even fail (e.g. Denial of Service attacks) when the number of request goes beyond its serving capacity. Applications in such situation could discard additional requests - a case of unavailability - to maintain response time and throughput, or suffer performance issues. In either case, the system has failed to scale properly.

One approach to scale an existing system is to increase system resources, especially the ones being bottlenecked. This approach is called *Vertical Scaling*, as it involves scaling up of resource capacities at a single node (such as application server). This solution is very simple and only requires investment in high performing high capacity hardware such as additional RAM modules, hard disks, network switches. An

application can also achieve better performance using multi-core processors. The advantage of vertical scaling is that it does not require changes in application and system architecture. Moreover in virtual box environments it allows proper provision of resources, such as dedicating a processor core per virtual operating system. However, high-end hardware is expensive and failure of any hardware component leaves the system unavailable.

2.4.1 Clustering

The single point of failure and other drawbacks of vertical scaling have lead to a fault-tolerant, highly available and scalable scheme known as *Horizontal Scaling*. In this approach, more nodes are added to a particular tier to scale it out horizontally, forming a *cluster*. As computer prices have dropped, it is very inexpensive to create large clusters by adding cheap commodity machines. In case of web applications, more nodes could be added to the application tier or the data tier or both. Figure 2.6 depicts a widely preferred architecture where only the application tier is clustered.

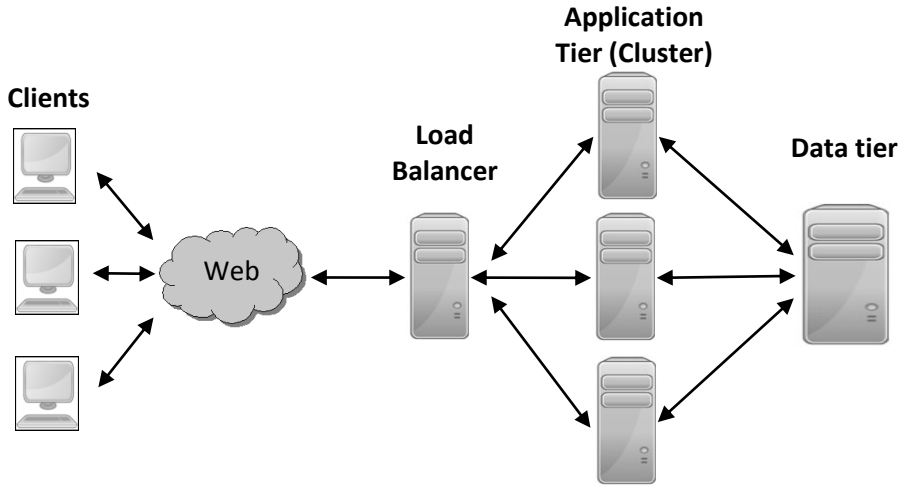


Figure 2.6: Clustering

Application Tier Clustering:

Clustering the application tier into several nodes (application server instances) provides the system the much needed scalability. It reduces load on individual nodes and increases system throughput. The entire cluster is meant to run the same application, and if one node fails the others can continue to provide service. As the cluster size grows the system becomes more fault-tolerant and available. One problem of clustering is that the application sometimes needs to be modified to take advantage of aggregate cluster resources. This is particularly true for distributed applications where one node communicates with other nodes, for example to replicate user sessions or application state. Because of the popularity of clustering application servers vendors now provide distributed session management capabilities.

Data Tier Clustering:

The data tier could also be clustered. However, this approach is rarely used as it requires RDBMS servers to actively keep their database state consistent with others. This also complicates transaction management and increases data integrity issues in the business tier as it will have to connect to many database servers. The other approach is to use one RDBMS server and several replicas of databases, of which one is the master (used for writing) while others are slaves (used for reading). This approach has its overhead too, as the master has to continuously replicate any changes to the slaves. If lazy replication is used then the slave state may get stale while eager replication inhibits request processing during the replication phase. Moreover, if the database size is big then replication increases disk requirements proportionally to the number of slave instances. Hence, many web applications refrain from both approaches.

2.4.2 Load Balancing

Another problem of clustering is that now a client needs to be aware of multiple servers instead of one. And whenever a node is added or removed the client somehow should know about it. To hide the intricacies of a clustered application tier a load balancer is introduced in between the cluster and the client, and the client merely needs to know the web address of the load balancer to whom it sends web requests. The task of the load balancer is to distribute client requests to different application servers and then return the response back to the client. The load balancer also sends heart beat messages to servers, so that it can identify failed nodes.

The load balancing algorithm could be implemented in hardware but software-based load balancers are prevalent because they are cheap and provide the flexibility to choose or even write new load balancing algorithms. The two broad categories of load balancing are described next.

Content-blind Approach:

In this case, the load balancer distributes requests without looking at the content of the request. However, the load balancer may consider system resources such as IO and CPU load to avoid bottlenecks at one or more server. The following content-blind approaches are very common and provided by almost all vendors

- **Round Robin:** The requests are distributed in round robin fashion, so the first request goes to node-1, the second request goes to node-2, and so on. Sometimes, nodes may have different hardware configurations and thus, different serving capacities. In this case, a weighted round-robin algorithm

could be used where a node with twice the weight of another node is selected twice as often as the other.

- **Least Connection:** The server with the least number of open connections is chosen. Again this could be weighted or not.

- **Least Loaded:** This approach periodically monitors each server for CPU, IO, and Network loads and selects the one with least load.

- **Random:** This is the naïve strategy where a server selection is random. Good pseudo numbers could provide uniform load distribution.

Content Aware Approach:

Contrary to content blind algorithms this approach peeks into requests in order to decide on a suitable server. For example, a cookie based load balancer would look at the ServerID cookie in the request header and would send it to that server. This cookie could be injected in the response header by the load balancer or the application server to create persistent connections. This approach is used when a user session is not replicated to all servers, and consecutive requests of the same client have to be served by the same server.

Content aware load balancing could also be application aware. For example [6] proposes an application-aware load balancing solution where different application transactions are predetermined and mapped to different servers. The request parameter contains the transaction type from which the load balancer can get the appropriate server to maximize cache hit. In this thesis we propose a request-aware but application-unaware load balancing solution.

2.4.3 Caching

One way to improve application performance is caching. Caching is a process that improves application performance by storing the data in a cache so that it can be retrieved comparatively faster in subsequent requests. The data to be cached is either the result of a complex computation (function) or a copy of original values read from another component. If the requested data is found in the cache it can be quickly served from the cache rather than fetching it from the original component. This constitutes a *Cache Hit*. If the data is not found in the cache we have a *Cache Miss*, and the data has to be recomputed or fetched again, and then stored in the cache (Cache Put). Figure 2.7 illustrates caching through a sequence diagram.

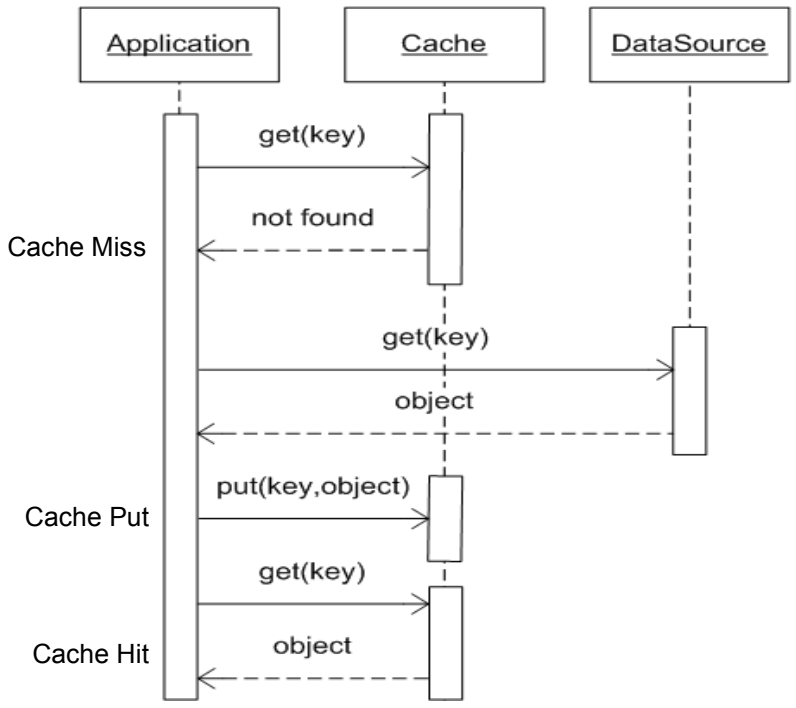


Figure 2.7: Caching sequence diagram

Each GET request is directed to the cache first, and if the key is not found in the cache then it is fetched from the data source. The object is put as key-value pair in the cache. The second request for the same object results in a cache hit. As evident from Figure 2.7 caching also reduces the work load on the original component as it bypasses it in case of cache hit (see second get request). Thus, if application scalability is restricted because of a bottlenecked component, placing a cache in front of it would reduce the bottleneck and allow scalability. The more a cache is able to serve requests the better the performance. The percentage of requests that can be served from the cache is called *hit ratio*.

Despite being a scarce resource the cache plays an important role in an application because of data access patterns. The data which is recently accessed is likely to be accessed again in the near future and constitutes temporal locality of reference. For example, product categories in an e-commerce website exhibit temporal locality as they are accessed in each request. Another type of reference locality is spatial locality where the data which is stored close to recently accessed data is likely to be accessed in the future. For example, a paginated list of items exhibits spatial locality as pages may be accessed in tandem. An intelligent application would identify spatial locality and prefetch related objects into the cache.

Caching has become an indispensible component in multi-tier web application. A dynamic request originating from the client browser has to go all its way through the web servers, the application servers, and the database server which takes considerable amount of time. Introducing appropriate caches along this route before any tier can decrease the response time. We now briefly discuss different types of caching in internet applications.

Page Caching at the Web Server:

Web servers can cache pages generated by application servers. This caching is for dynamic requests and is different from any browser caching which caches static html pages and images. Usually, the web server is part of the application server (such as in JBoss, Glassfish) but sometimes a high performing standalone web server (like Apache) is placed in front of the application server to serve static pages. In either case, some of the dynamically generated pages or page fragments (in a web portal) could by cached if these are not frequently updated. These pages are mapped to request URLs which may or may not include query parameters, and stored in the cache with an expiry after which they are regenerated. For example, weather reports for a city could be cached every 30 minutes. Thus, any request for this page would be served immediately by the web server saving trips to the application servers and backend data services. Apache uses several interlinked modules such as mod_cache, mod_disk_cache, mode_file_cache, etc, that can be added to the web server to enable page caching [7].

Object Caching at the Application Server:

Object caching is done in the application server, specifically at the data access layer. An object here represents a business entity which is loaded from the database server and stored in the application server cache. Having an entity cache not only improves processing at the application server but also spares the database of frequent requests. This is very effective if the application tier is clustered because nodes querying the database simultaneously increase the load on the database server. Under peak load the database becomes the bottleneck as requests starts to queue at its end, which indirectly impacts the application servers' performance as they are blocked on network IO.

If the application server does dynamic caching it has to deal with the volatility of data.

For many business applications the cache cannot afford to have stale entries, which are inconsistent with the database. A general approach to avoid such a scenario is to tag each cache entry with a time-to-live (TTL) and then the cache automatically evicts those objects upon expiration. If the database could be changed by another application then TTL based eviction is a good solution. In this thesis we primarily deal with object caching, therefore it is further discussed in Section 2.5.

Caching at the Database Server:

The database server, which is the backend of our Internet architecture, is very important for application performance. The application server, during its request processing, usually calls the database server more than once. The processing at the database server is also resource intensive and takes considerable time. To improve performance, database systems employ indexes to avoid sequential table scans and speed up queries. They also load blocks (or pages) of the database into memory buffers to reduce disk IO which also enables spatial locality of reference. Some databases, such as Oracle [8] and MySQL [9], also cache query result set to avoid query compilation and execution costs.

2.5 Caching in Java Application Servers

We focus in this section on object caching in Java web applications as our implementation is based on it. However, the concepts presented here are very general and applicable to other frameworks as well. As discussed in Section 2.3, Java application servers support ORM frameworks such as JPA, Hibernate and Toplink. These frameworks have the capability to cache database rows as entities in the application server. The entity caching is done at two levels: session and application, the architecture of which is shown in Figure 2.8.

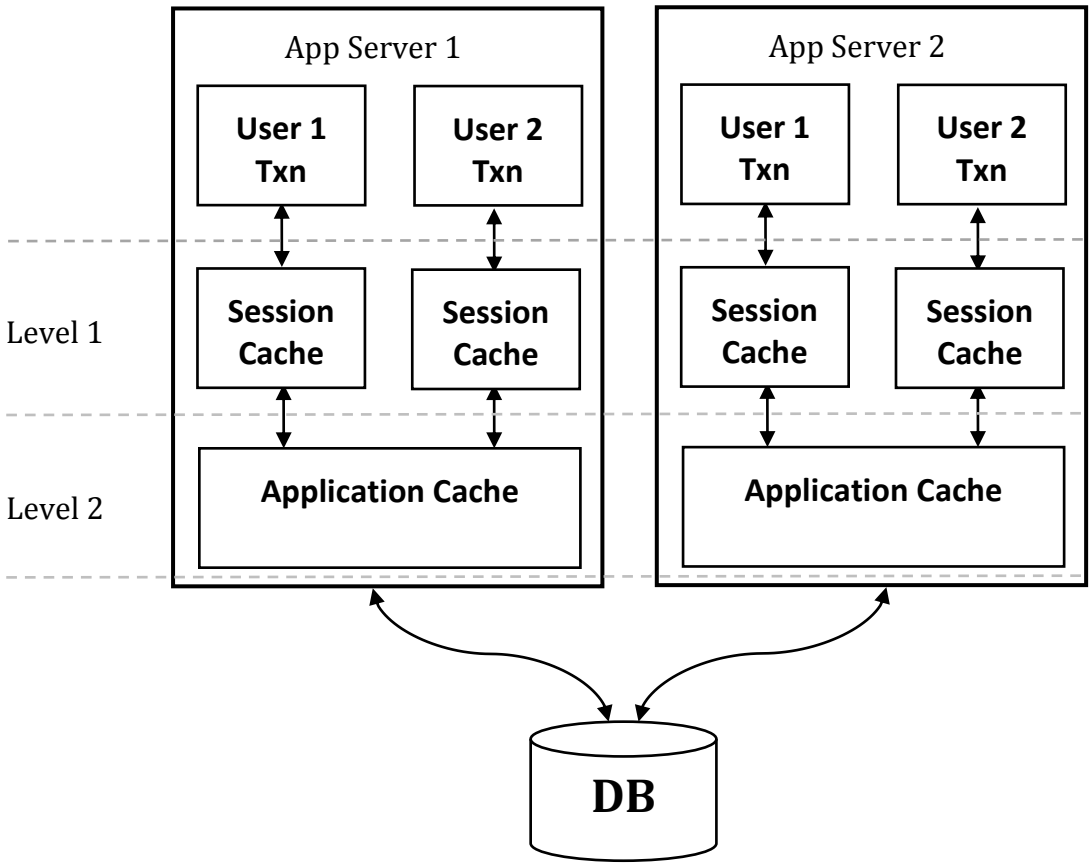


Figure 2.8: 2 Level caching architecture

Session Cache:

Session Cache is the first level cache and has a request scope. When an application server begins a database transaction for a particular user request, Hibernate creates a session level cache. Any query executed during this transaction is temporarily cached in this session cache until the transaction is over. This cache is created per user transaction and is mutually exclusive from any other transaction running concurrently in the application server, and hence, it is sometimes referred to as transactional cache. However, depending on the ORM framework used, this can also persist at thread level

where multiple transactions (executed serially) all share this cache. This cache is embedded in the ORM framework and generally cannot to be disabled as it buffers user modifications and batches them for final commit.

Application Cache:

Application Cache is the 2nd level cache and has application scope. The key characteristic of the second-level cache is that it is used across sessions, which differentiates it from the 1st level cache. Just like other multi-level caches, the 2nd level cache has more capacity than the 1st level caches. Hibernate reads from the application cache if there is a miss in the session level cache, and writes (puts) to the application cache after querying the database. Any intermediate changes to the entities during transaction are not propagated to the application cache; however, final updates to database are (write-through approach). Because of its global scope any transaction - be it from the same user or others - has access to this cache. The application cache does not span multiple servers and each server has its own application cache which limits cacheability of the system.

Hibernate provides a flexible concept to exchange cache providers for the second-level cache. We have used Ehcache [10] as cache provider. However other caching implementation like OS Cache, Swarm Cache or JBoss Cache can be used.

Query Cache:

In addition to entities, the 2nd level cache also keeps recently executed queries in a separate cache called a Query Cache. It is treated slightly differently than the entity cache by the framework, although the storage mechanism is exactly the same as the entity cache. When a query is executed it is stored in the query cache as <query, result set> pair. The result set only contains the identifiers (primary keys) of the fetched

records. Hibernate then queries each entity in the result set and stores them in the entity cache. In order to use the query cache with Hibernate each query that needs to be cached has to be explicitly marked cacheable.

2.5.1 Distributed Application Cache

Even though the application level cache improves performance of the system it does not help to scale the system in a clustered environment. Each application cache is oblivious of other application caches in the cluster. Hence, there is a high probability of redundancy in all the caches. This probability is much higher if the load balancer distributes requests in a round-robin fashion as each application server will be caching the same entities. This redundancy entails the following critical problems:

- Since application level cache is usually in-memory it is limited, and redundancy reduces the overall cache capacity of the system. Hence if each of the N nodes has 1 GB of available memory, then the utilization of the entire cache would be much less than N GB because of duplicates.

- The application caches fill up quickly as each individual server has access to its own limited cache. Now, if a request is sent to a server which does not have the corresponding objects cached while other server have, then it will still be fetched from the database. This not only results in a database hit but also in the eviction of some entities from the application server cache which may be queried next resulting in further evictions.

- Cache redundancy can lead to inconsistency. If one node inserts or updates a row in database, its cache would be updated while other nodes would contain a stale cache entry. Requests for these objects on other servers would be

served from the cache which does not represent the data's current state. Many financial and e-commerce application cannot afford this anomaly.

All of these problems can be solved using an in-memory distributed application cache which exploits the memory capacities of all servers to give a transparent view to the application. A distributed caching layer conceals access to and from other application caches. If the data is not found in the local cache it is fetched from a remote cache rather than the database. Pinpointing the location of a remote cached object can be done either by maintaining a cache directory at each server or using consistent hashing. The cache directory is replicated across the cluster and contains the mapping of cache keys to server locations. In consistent hashing, cache keys are hashed (using modulo) to server ids.

Distributed caching is getting popular because an application can simply be scaled by adding cheap commodity machines and memory. Also, the network cards and switches are getting faster, with 1 GB the norm while 10 GB getting traction allowing low latency remote fetching. Some of the well-known distributed cache frameworks are further discussed in the following related work section.

2.6 Related Work

A lot of research effort has been put to improve the performance of web applications and make them scalable. While some approaches concentrate more on scalability than performance, we have tried to tackle both issues in this research by allowing users to choose from various distribution and replication strategies that fit best for their application-hardware configuration. Our approach is more holistic in a sense that it analyzes the application workload and applies policy based tuning at the load balancer and application tier.

Some of the existing research relies on offline simulation and benchmarking of application to derive useful metrics for system tuning. For example, in [11] and [12] the authors measure performance characteristics of different tiers in an Internet application, and compare predicted performance with test bed application performance. They extended their model to capture load imbalances using replication, caching and resource provisioning. The motivating factor of their research was an experiment where they were able to predict the response time within 95% confidence intervals of the observed response time. Similar studies in [13] and [14] assert that queuing models can be very useful in measuring and predicting performance characteristics for 3-tier web applications. In this model, they specifically discuss the architecture of each tier on prevailing open-source servers, and were able to demonstrate how precisely analytical models could mimic real time performance. Our analysis is differentiated from them in that it is not based on offline queuing models but rather on live application feed. We contend that application workload may change and it is best to adapt to varying workloads periodically without taking applications offline.

Instead of analytical modeling, some researchers have forayed into different strategies, such as caching and replication, to scale web applications [15]. For example, content delivery networks like Akamai employ Edge Computing to reduce latency by replicating application logic and part of the database on edge caches. In [15], the authors also evaluate content-aware and content-blind caching strategies on edge servers and conclude that no approach is suitable for every workload. They put the onus on administrators to manually choose a desired caching/replication strategy for their workload. Although we do not deal with performance tuning on the edge, however, we do focus on latency reduction using content-aware request distribution. Moreover, our framework reduces complexity involved in manual configuration by

exposing only required tuning parameters (such as cache replication factor) to the administrators.

Distributed caching in web application holds a key to performance and scalability, as discussed earlier. This problem was addressed by Memcached [16] which is a distributed memory cache that exploits available memory on any node by running a Memcached instance on it. Memcached offers a dictionary interface where the cache key is first hashed to identify the servers and then hashed again to retrieve the object. Memcached was originally implemented for LiveJournal.com with 30GB of aggregate memory, which resulted in 92% cache hit rate. Memcached servers run separately as different processes. Hence accessing them is slower than accessing an embedded cache in the application server. And because of key hashing, Memcached is purely distributed and does not allow for replication. Also, no control is given on what to store where. Similar to Memcached is Terracotta Server Array (TSA) [17] which acts as a distributed level 3 cache in web applications (level 2 in non-JPA environment). The 2nd level cache in the web application replicates 100% of data, which limits scalability but provides data consistency and cache coherency.

In [18], a distributed cooperative cache is built on top of EhCache which runs in the application JVM. Its distributed directory maps cache keys to other application server, which, unlike hashing in Memcached, avoids unnecessary remote lookups if the entity is not even cached. It is also more flexible than Memcached in regard to where to put each object. In our implementation we have enhanced the distributed cache to support replication, so that popular objects could be retrieved much faster from multiple application servers sacrificing some cache space for performance.

Content-aware load balancing presented in [19] logically partitions web content into different servers, and creates policies for accessing them at the load balancer. This

load balancer is thus content aware, and sends a request to the server that has a high probability of hitting the cache. Similarly, authors in [20] propose memory-aware load balancing where a transaction is sent to a particular replica that has the required working set to execute it completely in memory. In general, by localizing requests to a given server the caches can now hold on to partitioned content much longer by avoiding frequent evictions. In [6], the author takes this approach to dynamic database content by logically partitioning different types of transactions and assigning them to servers. Similarly to [19], the load balancer sends request of one type to the same server, thus maximizing cache hit rate. In [6], the web application has to be manually profiled to identify all types of web transactions, and a static policy file is generated for the load balancer. Researchers in [21] and [22] argue on separating architectural concerns, like caching, from the implementation of internet services. They show that some level of automation could be incorporated into multi-component web services. They analyze bottlenecks along a chain of linked service components and rewire them with appropriate cache capacities. We follow the same level of automation in our infrastructure but at the macro level of application tiers. Our system can trace requests and their resources for any web application on fly, which removes the need for manual profiling. This technique also aids us in generating content-aware but application-unaware distribution strategies.

Chapter 3

Holistic Caching Architecture

3.1 Introduction

The performance of any software application relies heavily on its hardware infrastructure and software architecture. It is important that application components are loosely coupled with extensibility support. Keeping this in mind we have developed a holistic caching architecture that is robust and adds no overhead to the actual application but rather improves it end-to-end. The first step to achieve holistic caching is to augment and enhance the infrastructure of the 3-tier architecture. Although the request processing still follows the classical path (i.e., load balancer to application server to database server), we have added another server for runtime statistics collection and processing. This server should ideally be on a separate machine to avoid resource stealing from the main servers. The web application is oblivious of this additional server. Our main objective is to keep the application as-is and unaware of optimizations. This allows us to build one optimization framework that is readily applicable to other applications. Figure 3.1 shows our infrastructure with its high level components.

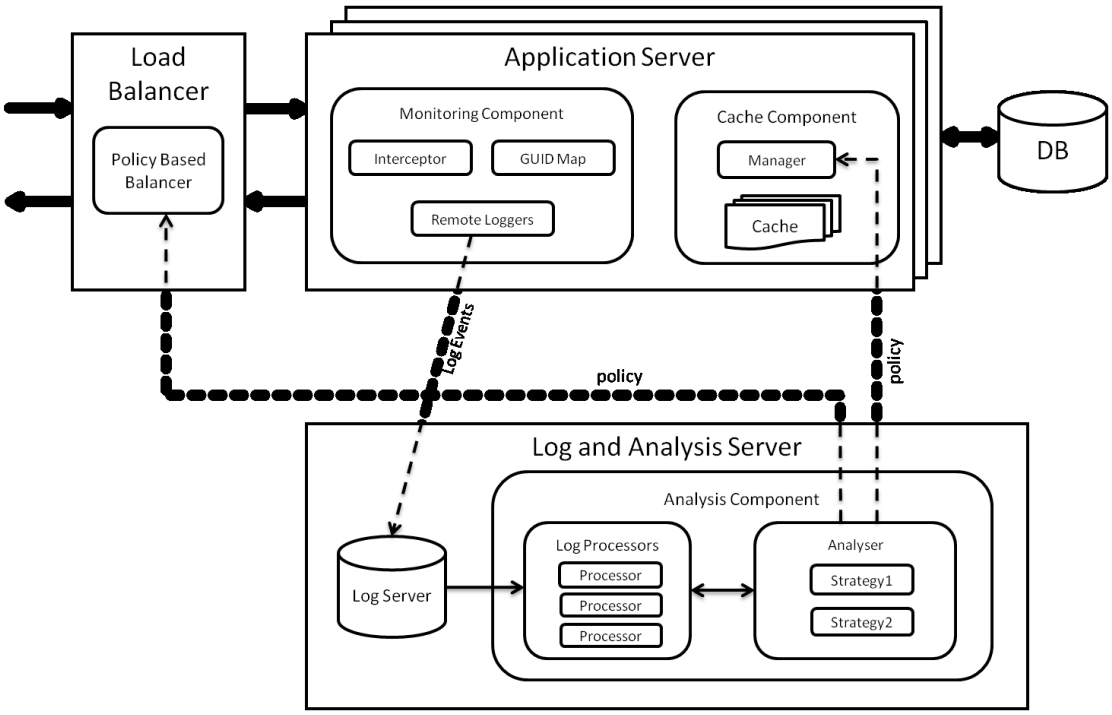


Figure 3.1: Holistic caching component diagram

In a nutshell, to perform optimization we need to gather runtime statistics which are mainly collected from the application servers at two critical components, the request/response interceptor (i.e. the filter) and the cache component. These components log fine grain information to a separate log server. The log server receives logs from all application servers and processes it through an analytics engine which will generate optimization policies for request distribution and cache replication and distribution.

In the next section we briefly discuss the distributed cooperative cache presented in [18], and our enhancements to it. Next we give an overview of how our *Policy-Based Request Distribution and Caching* transparently improves application performance. The remaining sections discusses major component of our system and their interactions. Part of this framework was built in collaboration with S. Joshipura [23].

3.2 Holistic Cache Component

Clustering application servers also clusters application server caches. Since caches in the cluster are totally unaware of each other, there is a high probability that at a given point in time there are duplicate cached objects in the cache cluster. This redundancy not only leads to cache inconsistency but also affects application scalability. The impact is more if caching is done only in main memory, which is an expensive hardware with limited capacity.

The application performance can be easily improved if all these distributed caches somehow add up to give a uniform single large cache. To achieve this, caches have to be aware of each other, so as to not cache things already cached on other servers. Moreover, they should be able to retrieve objects stored on remote caches. This scheme not only removes redundancy but also increases cache hit rate because of overall increased capacity, which further reduces the load on the database.

This collaborative caching termed as 'Distributed Cooperative Cache' in [18] is achieved by way of a cache directory which knows the whereabouts of each cached object in the whole cluster. The cache directory is replicated across all application servers and kept consistent as cache objects are stored and removed from the cache. When an application server needs an object not found in its local cache, the cache directory resolves the cached object's location from the cache key, and hence, the object is fetched from the remote cache.

Our work is build upon this cooperative cache component which uses EhCache as an underlying application level cache. We have done a few enhancements to this caching component essential to achieve better caching, such as:

Object Replication Support:

The cache directory now supports object replication. That is, if an object X is replicated on N servers to improve availability, all cache directory replicas contain the list of these N servers for this object X. Any request for object X can now be served from any of these N servers if not found locally. This redundancy not only adds fault tolerance and availability to the caching system but also increases local cache hits at the cost of cache space.

As discussed in the last chapter, replication creates consistency issues when there are updates. Consistency could be achieved by invalidation where updated objects could be invalidated and thus, removed from the cache. But since our focus in this thesis is on read-only workload we have not implemented consistency mechanisms.

Singleton Cache instead of Disjoint Region Based Cache:

The default implementation of EhCache (for Hibernate) creates unique caches per database table called *Region Caches*. Thus, if there are hundreds of tables in a database then there will be as many cache instances. Each of these cache instances - usually named after the table - only stores objects of its associated table. This approach, although if achieves better concurrency for update heavy traffic, has obvious disadvantages such as

- Each cache region has to be specifically configured and adjusted on constant basis. For example, when a certain table grows or its access frequency increases then the system administrator needs to increase the cache capacity for this region and perhaps penalize some other region caches by decreasing their capacity. Performing such manual tuning on all application servers is very time consuming. Not doing so could lead to cache inefficacy.

- Not only objects are stored and retrieved from an associated region cache, but also evicted from the same cache instance. This has two undesirable effects. First, the cache eviction policy is forced to choose an object for eviction from this cache when its capacity reaches its limit, even though other region caches have free space. This puts a hard cap on region caches. Second, the eviction policies can only evict from the given cache (for which there is a put request). Hence, eviction policies such as LRU are oblivious of least recently used objects in other region caches (than the candidate chosen from this cache). Finally, not only the inactive region caches waste valuable cache space but also increases thrashing for popular regions.

To overcome all these drawbacks we have created a singleton cache shared by all regions. When Hibernate requests for a region cache the same singleton instance is returned. This makes it easier to setup cache capacity based on available memory, gives eviction policies full view of the application cache, and makes automation tool like ours to generate cache policies effectively.

Popularity Aware Eviction Policy:

The LRU eviction policy works best most of the time. However, web requests follow patterns where there is a likelihood that a naïve LRU policy would evict an object blindly which is about to be requested. This is particularly true when cache capacity is small compared to the database size, and many random requests interleave popular ones.

To alleviate this phenomenon we have introduced cache object stickiness implemented through a *Time-To-Live (TTL)* cached object property. TTL is the time after which the object expires from the cache and becomes a candidate for eviction. By increasing the TTL for popular objects we reduce their chances of eviction, thereby

letting them *stick* to the cache a bit longer than less popular objects. We propose three different levels of TTL:

- ***Short TTL:*** This is the base TTL set for any object in the cache. Objects are very rarely accessed within a timeframe should be assigned short TTL. Objects with short TTL are more likely to be evicted from the cache.

- ***Long TTL:*** Long TTL has a longer life (TTL value) than short TTL but still the object could be evicted if it is not accessed for a long time, thus evicted through expiration or as an eviction candidate. Popular objects could be cached with Long TTL.

- ***Stable TTL:*** Objects which are always meant to be kept in the cache are stored with a stable TTL. Depending on the application, stable TTL could be set in terms of hours or even days. Examples of objects to be stored with stable TTL are dictionary tables and catalogues that are accessed extremely frequently.

These little refinements on the existing open-source software give us the flexibility to better tune the application for various workloads. In the next section we describe the fundamental components of our infrastructure, the *Policies*, and how and where do they fit in our distributed multi-tier web application.

3.2 Policy-Based Request Distribution and Caching

Expert Systems are software that mimics human reasoning to solve a problem. A more prevalent type of expert system known as *Rule-Based Expert System* constructs a set of procedural rules to solve a problem. The *rule* here is a trigger that fires when a given condition is met, to partially or fully solve a problem. Having a large set of rules not only increases rule trigger-rate, but also allows an expert system to come up with

the best possible solution. Many scientific and diagnostic systems are built as rule-based expert systems.

Our proposition is similar to a rule-based system, where rules are created to efficiently serve a user request. We call our rules *policies* and hence, our approach 'policy-based' approach. It is also important to understand the distinction between a strict rule-based approach and our policy-based approach. In a strict rule based system, requests (inputs) are ignored if none of the rules are able to comprehend it. Contrary to this our policy-based approach is assistive and improves system performance if the policy is applicable to the user request. If it is not, then, the default system functionality is performed and we still achieve a response, but perhaps not in the optimal way. Thus, our goal is to maximize policy hit-rate, which in turns increases system performance and throughput.

Our intention is to improve system performance by offloading the database. We do this by increasing the cache hit-rate, especially the local cache hit-rate. Having a distributed clustered cache offloads the database considerably. It does not, however, necessarily improve response time. This is because a remote cache request involves network latency. A random or even a round robin distribution of user requests to different application server spreads objects all over our cache cluster. Therefore, a request which might be accessing merely 3 objects, each from a different remote cache, could be much slower than one accessing 30 objects from a local cache.

To alleviate this problem inherent in a blind distributive cache, we silently profile application requests and resources, and observe access patterns in them. As the web application runs and users start querying the servers, we collect all this information on a continuous basis. The collected requests and resources, their associated data, and their mappings collectively form our knowledge base. We then analyse this structured

data set and generate request distribution policies for the load balancer and the corresponding cache policies for the application servers. These policies are defined next.

3.2.1 Policy-based Load Balancing

As discussed in the last chapter, basic load balancing schemes are content-blind - that is, they distribute requests across a cluster without looking at the request. Our approach to load balancing is content-aware in that it peeks at the request and knows in advance where this request would be served faster. This distribution by the load balancer takes advantage of the cache present on the application servers. For example, when a dynamic request X is sent to application server AS1, then the object accessed by it would get cached on this server so that subsequent requests are served immediately without going to the database. Since our load balancer is content-aware it would send the next request X to AS1 which would then be served from the cache. This sticky distribution is achieved through *Load Balancer Policies*. These policies are generated by the analytics server and the load balancing algorithm takes advantage of these policies and performs efficient distribution.

The load balancer policy is a collection of request to application server mappings. The structure of this policy is very simple as shown below.

www.example.com/req1/	**->**	**{Application Server 2}**
www.example.com/req1?id=1	**->**	**{Application Server 1}**
www.example.com/req1?id=2	**->**	**{Application Server 3}**
www.example.com/req2	**->**	**{Application Server 1, 2}**

The first mapping maps *req1* to application server 2. The second and third mappings

are for the same request but with different parameters (a.k.a. Query String). Hence, these are treated as two different requests and can be mapped to two different servers. This brings us to the concept of URL processing in an application server. In web applications, all the first three requests are served by the same server side script *req1* (or an equivalent *Servlet* in Java-based application servers). The server side script may in fact behave differently and access different resources based on query parameters. Therefore, we incorporate fine grained URLs in our mappings. Although this increases the size of our structure, the load balancing algorithm works upon a large set of URLs and achieves better and uniform request distribution.

As evident from the last mapping, one request can be mapped to multiple servers, in which case the load balancer would send the request to any one of these server. The provision to have a single request being served by different servers opens up dynamic replication possibilities on application servers provided each server locally caches the objects it accesses rather than fetching it from a remote location. Under peak workload, where a few requests are frequently accessed, mapping them to only one server could overload it. This multiple server configuration helps us balance load on each server.

3.2.1 Policy-based Caching on Application Servers

To efficiently utilize the distributed cache we generate caching policies which work well in tandem with our load balancer policies. As discussed in Section 3.2 we have devised a new mechanism to caching where objects could be cached with different TTL. This enables our analytical engine to assign different life time to each object based on its popularity. Moreover, a clustered distributed cache needs a tighter control over what objects are cached and what are not when they are fetched from a local cache or a remote cache.

Before describing different caching scenarios in our holistic cache environment, it is imperative to understand a typical cache policy. A cache policy looks like

com.example.User#100 -> { Server 1 (Long TTL) }
com.example.User#200 -> { Server 1 (Long TTL), Server 2 (Long TTL) }
com.example.Country#Canada -> { Server 1 (Stable TTL), Server 2 (Long TTL) }
com.example.Country#USA -> { Server 3 (Short TTL) }

Here 'com.example.User' is the ORM entity for the database table User, and '100' is the identity of the object representing the primary key of the corresponding row in the User table. Thus, *User 100* is destined to be cached on application server 1 with a Long TTL. Analogous to the load balancer policy, objects could be cached on different servers with same or different TTL values, again giving the analytics server flexibility to create complex cache assignments. For example, during the Winter Olympics in Vancouver, a website could encounter more traffic for Canada, and may increase the priority of Entity-Canada to Stable on one server and replicate it as Long on another, while reducing that of Entity-USA to short.

The cache policies are created for individual application servers and may include duplicate entries to support object replication. Thus, the above cache assignment translates to this equivalent server-to-objects cache policy.

Server 1 -> { User#100 (Long TTL), User#200 (Long TTL), Country#Canada (Stable TTL) }
Server 2 -> { User#200 (Long TTL), Country#Canada (Long TTL) }
Server 3 -> { Country#USA (Short TTL) }

3.2.1 Caching Scenarios

The flow chart below illustrates how the cache components perform when an object 'Obj' is requested by a data access layer (such as Hibernate).

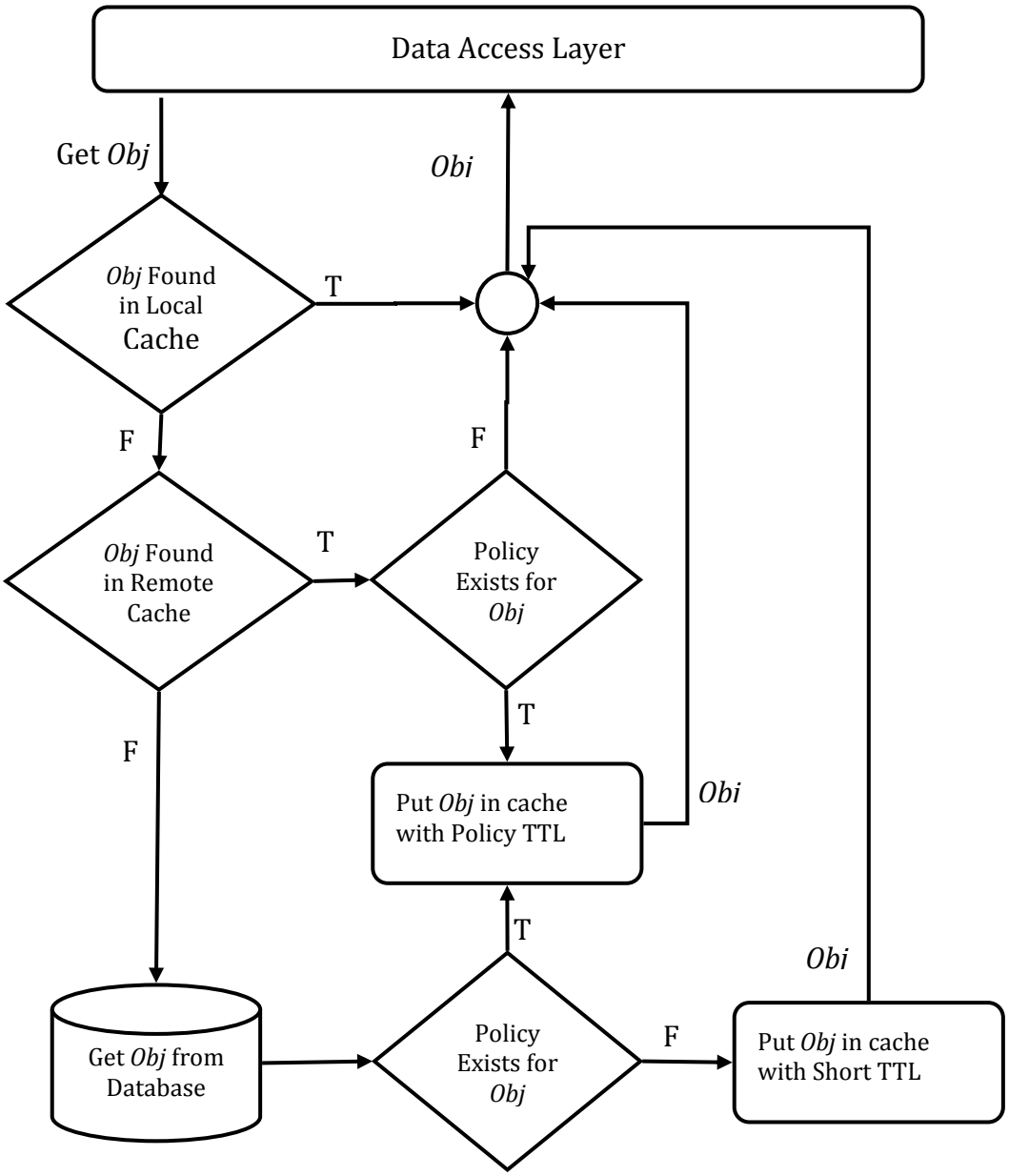


Figure 3.2: Caching scenarios

The GET request for an object is intercepted by our cache wrapper which first checks the local cache for the object. If it is found there, it is immediately returned. Otherwise, we look into our cache directory for a remote presence. If the object is found in a remote cache it is retrieved from there. Depending on whether a policy exists at the local application server for this newly retrieved object it is stored in the local cache with a policy-based TTL, else, the object is returned without being put in the local cache. This allows us to only cache objects from remote caches which are meant to be replicated according to the policies generated by the analytics engine. If the object is not found in the clustered wide application cache, then it is fetched from the database and put into the local cache. Again, if the policy exists for this object it is stored with a given TTL otherwise with the base short TTL. The directory structure is kept consistent across all servers when objects are stored and evicted from the cache.

3.3 Monitoring System

We have developed a monitoring system that extracts useful information from the user's HTTP requests and the cache accesses. We, however, do not monitor application business logic and internal communication which allows our monitoring system to be deployed along with pre compiled web applications. This is a really important aspect of any automation tool where non-invasive components are seamlessly integrated with a functional system. These components can also be detached from the application with ease, which makes it ideal to do application profiling and data mining for experimentation purposes. However, we have aimed to keep the overhead of monitoring to a minimum so that the system performs equally well with the additional monitoring layer. Now, we discuss high level concepts about our monitoring system, the implementation aspects of which are presented in Chapter 5.

3.3.1 Remote Logging

The crucial aspect of any monitoring solution is logging of information. An inefficient logging could sabotage the whole monitoring system. Keeping in view our need for dynamic policy generation, we chose remote server logging over traditional local file system logging. Although File based-logging is robust, in a clustered environment it does not prove to be feasible for the following two reasons.

- Each server logs records in its local disk, which then need to be merged based on timestamps or other mechanisms to get the global view of application logs. This usually cannot be done for active log files as it's been constantly written to.

- The processing of file logs is resource intensive and cannot be performed on the application servers. Also, in this arrangement, one application server needs to be a master log processor reading files from other servers which could hog the network during log processing cycles.

Remote logging addresses both concerns seamlessly. By virtue of remote logging we are able to output the log directly at the log server, a separate machine in the server cluster, where the analysis tool is running. Thus, the analysis tool is fed with the logging data in real-time as it is produced. This makes the policy generation more dynamic. Moreover, the raw log events are never materialized to the file system saving valuable IO cost.

3.3.2 Request Interception

Requests are intercepted using the *Interceptor Pattern.* According to [24]

"In the field of software development, an interceptor pattern is a software design pattern that is used when software systems or frameworks want to offer a way to change, or augment, their usual processing cycle"

In web applications, we just augment the usual request processing by adding interceptors to collect request statistics. This interceptor is really fast as it does not change requests or its processing, but merely monitors and logs http access details. A sample entry in an access log looks like

REQ 1233453464 GET /rubis/cateogries?id=10 200 150

The first value identifies the log type which is the request. The second value is the timestamp of the incoming request. The third value is the HTTP method. The fourth value is the requested resource along with any query string. The fifth value is the response code and lastly the total time taken in millisecond for this request to complete.

3.3.3 Cache Monitoring

We monitor cache accesses analogous to HTTP requests. Both GET and PUT accesses are logged. A log entry for a GET request contains the timestamp, request type which is GET, and the object key. The key is composed of Entity Type and its primary key. A sample log entry is shown below

1233453464 GET com.example.User#100

The PUT request is made after an object is fetched from the database. Usually a GET request precedes a PUT but if the data access layer is sure that the object cannot be in

the cache, such as for Inserts and Update queries, a 2nd level cache PUT is done. The PUT log entry is similar to GET with an additional object's size parameter, and it is shown below

1233453464 PUT com.example.User#100 300

It is worth mentioning that we do not log the objects value here, because a value is usually big. This would slow down the remote logging considerably. Also, for application unaware tools, like ours, performing analysis on values is much harder. Therefore, we only store object size, which is a reasonably accurate estimate on entity's caching and network bandwidth needs.

3.3.4 Request Tracing

Request tracing is the most important part of our analysis. Simply put, *request tracing* is a way to identify all objects accessed by a request. In web applications, the data access layer is oblivious of web requests and is only concerned with data objects or entities. Thus, to overcome this barrier, we include a unique global id (GUID) with each log message sent. One GUID is created per thread, and since a request is processed in one thread all cache logs get the same GUID as sent along with the request log. This is essential because as the log server receives interleaved logs from concurrent request, a GUID is needed to recollect and identify them as part of one request. This helps us to determine data access patterns of requests. Furthermore, using this information we can direct the load balancer to send specific requests to the application server that contains the cache objects so that local cache hits are maximized. In addition to GUID, we also send the host machine name with each log message to distinguish logs from different servers.

3.4 Log Processing

As logs arrive at the Logging server, they are processed by different processors. We called them 'Log Processors' and their purpose is to collect log messages of interest and structure them in a meaningful way. Log messages from different servers are maintained in separate structures. This gives us the flexibility to analyse load on different servers. But for a global view of application we also merge them into one structure. The data structures produced by log processors are then fed to the analysis engine which then generates policies.

3.4.1 Request Log Processing

HTTP access logs are processed by the Request Log Processor. This processor identifies popular requests by their access frequency. In addition, it also maintains the first and last access time for each different type of request, and the average time taken for this request. All these structures are maintained in time buckets (say 10 min each), so that during analysis we can pick the last X buckets and merge them, to observe more recent trends and discard older analysis.

3.4.2 Cache Log Processing

Cache log events are processed by the Cache Log Processor. The cache objects are maintained in their lowest granularity, i.e., at the database row level. These objects can also be accessed at higher granularity (at table level) by just dropping the primary key from the cache key. For each object, we maintain its frequency, the number of GETs and PUTs, and first and last access time. Again, data structures are segmented into equally spaced time buckets.

3.4.3 Request-Cache mappings

One log processor in our framework maps cache objects to request. It processes both the request log and cache log streams and ties them together on the basis of GUID. Sample *Request-to-Objects* mappings are like

/req1 -> { a, b }

/req2 -> { b, e }

/req3 -> { x }

If the same request is encountered again during processing with different objects, then the previous mapping is updated with new objects, as shown below

/req1 -> { a, c, x} => /req1 -> { a, b, c, x }

Thus, we only store one mapping for the request as union of all objects. This allows us to identify a set of all possible objects that can be accessed by a given request. We also generate inverse mappings during processing from object to requests, enabling us to see object overlaps in different requests. The *Object-to-Requests* for the first 3 mappings are shown below

a -> { /req1 }

b -> { /req1, /req2 }

e -> { /req2 }

x -> { /req3 }

3.5 Analysis Engine

The analysis engine is the brain of our system. It contains the algorithms and procedures which generate the policies for the application server and the load balancer. It uses different statistics which have been extracted from the log processors to make decisions about the policies. The analysis is not performed continuously as log processing is, but at preconfigured time intervals. Since this is little application dependent, the system administrator can choose an interval on the basis of the workload.

We have kept our analysis engine as open as possible so different algorithms could be configured with different workload and application characteristics. The algorithms which we call *Strategies* work on the same set of structures, but use different heuristics and parameters to come up with unique request distributions and cache assignments. The generated policies are sent to all the application servers and the load balancer where they are applied instantly without requiring them to be restarted.

Policies are usually generated for both load balancer and application server, but sometimes generating one and not the other is sufficient to improve performance. Since policies are generated and applied at regular time intervals we are better able to manage variations in workload than a static configuration.

Chapter 4

Dynamic Request Centric Analysis

4.1 Introduction

This chapter focuses on core algorithms and strategies that enable us to improve application performance and overall system throughput. Here, we discuss a broad range of analysis on various factors to generate load distribution and caching policies for our multi-tier web application system. The caching subsystem of our multi-tier infrastructure is a distributed one. One of the important reasons of implementing a dynamic request centric caching and load balancing algorithm on top of a distributed cache is to minimize remote calls, both to the remote cache and backend database.

Our analysis is performed on metrics collected from the application server and the distributed cache. The distributed cache in consideration here is the persistent in-memory application level cache. In our analysis, we do not consider the session level cache, and neither have we done any monitoring at this level.

Our solution is implemented on Java Enterprise Edition (Java EE) application servers

and Apache Http Server because of its worldwide acceptance and open-source production quality implementation. Nonetheless, this does not restrict our approach to only Java based application servers. The whole infrastructure could be ported to any web application platform with minimal changes. This was another reason to keep the analytics engine separate from the application server; the policies generated by the analytics engine could be serialized into XML or another network friendly protocol with ease, making it a plug-and-play solution.

The analytical engine, although an add-on, then becomes the integral part of the whole system. Figure 4.1 shows the system with analysis engine, and the essential data input and output.

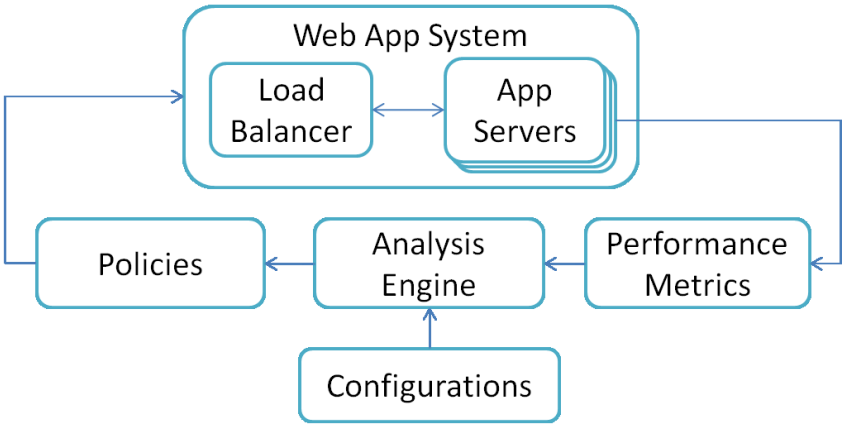


Figure 4.1: Analysis Engine

The data flow diagram shows how the analysis engine processes the performance metrics collected from the application server cluster and produces policies. The configurations are application specific properties such as cluster size and cache capacity, and algorithm tuning properties such as replication factor. The diagram does not refer to particular policy but shows *Policies* in general as an output. This is because the policies generated by the analysis engine effect the web application as a whole, even though in some cases a desirable performance could be achieved by applying

policies at either load balancer or application servers (see Section 4.8).

In this thesis we pursue a dynamic request centric analysis. The nature of our analysis is dynamic because it is done online with running servers, albeit on a dedicated machine, and the analysis adapts itself to changing workload. Our focus is on application unaware but request aware analysis (i.e. our analysis is oblivious of business application but aware of user requests). We term it a *Request Centric Analysis.* Having said that, we do consider cache statistics in our analysis, and we have used request tracing abilities of our system to build more controlled, uniform, and efficient request distribution policies. The cache policy more often plays an adjunct role to the load balancer policy because the default caching behavior of our system is distributed which minimizes backend database queries. The request distribution policy further improves on the default system by maximizing local cache hit thereby avoiding cache indirection. By virtue of this, we reduce network latency involved in fetching remote objects, thus improving response time. By localizing requests to servers, we virtually partition our cached data on different servers based on application needs, thus maximizing cacheability without incurring the network IO cost involved in remote fetching.

The rest of this chapter is organized as follows. In the next section we describe the important data structures and configurations which are used in nearly all algorithms. After that we discuss our base distribution and caching algorithm. Then we extend it to support replication. Finally we present a more complex assignment strategy that reduces cache redundancy.

4.2 Data Structures and Configurations

In our thesis, algorithms rely on performance metrics and data structures processed

by log processors as discussed in last chapter. These data structures are maintained separately for each application server, and even further segmented into time intervals. The first thing our analyser does is to merge each individual data structure taking into account only the more recent segments. Choosing the appropriate segment size is the key; in our approach, we neither disregard all structures processed in the last policy generation phase nor include all. Instead we include few segments from the last run. Once all these data structures are merged we feed them to our algorithm (aka Strategy). Below we outline the structures that are most important and used by every strategy.

- **HTTP Request List:**

  Since our analysis is request centric we maintain a list of all requests that access at least one resource. We only consider cache objects as resources. Thus, requests which access static html pages or images are not processed and are ignored. This is desirable as static resources are usually served faster. Moreover, not apportioning these requests gives the load balancer the opportunity to distribute such requests as evenly as possible. We also discard requests which are erroneous (whose responses are not OK; HTTP 200). The http request lists maintained for each server is merged, and then sorted on popularity. Our metric to identify request popularity is request frequency, which is a very good estimate. The list goes from most popular request to least one (sorted descendingly).

- **Request to Resource Map**

  All accessed objects are mapped to their requests at each server. These mapping are then merged from all servers to obtain an application wide *Request-to-Resource Map*. We need this structure so that we can create cache policies for the most frequent requests.

In addition to the prepared structures, the algorithms depend on the following basic parameters

- **CacheSize:** Capacity per server
- **NumServers:** Number of application servers
- **TotalCapacity:** CacheSize * NumServers

4.3 Basic Distribution Algorithm

Our first algorithm, the Basic Distribution Algorithm, addresses the following concern in multi-tier applications:

- A distributed caching system can relieve the database server from constant barrage of requests, by serving some requests from the local cache or a remote cache. However, the network cost of retrieving an object from a remote cache increases response time.

To minimize remote calls our first algorithm applies the following reasoning:

- Identify the popular requests which are accessing the database (or cache) most frequently.
- Assign a subset of these popular requests to specific application servers. By partitioning requests to server we maximize local cache hit rate.
- Allocate the objects accessed by theses requests to the corresponding server with a longer TTL.

We implement this simple policy by considering certain workload characteristics. First, the observed pattern is not always going to repeat, thus we cannot force all popular objects into the cache. Hence, we only consider the most popular ones depending on aggregate cache capacity. Second, we keep a portion of the cache unallocated so

objects for other requests could be cached. To do so, we put a cap on the number of cache assignments per application server. We also put a cap on total cache assignments by similar proportions. Figure 4.2 shows the algorithm setup while Figure 4.3 depicts the algorithm.

The basic distribution approach simply pops the most popular request from the list (line 15) and maps it onto one application server (line 24). It also generates cache policies for this application server (line 28-32) so the objects accessed by this request (line 18) stick longer in cache than less popular objects. The next popular request is then mapped to a different application server. Application servers are chosen in a round robin fashion (line 40). We continue generating policies until either the request list is exhausted, or we have reached the threshold set for popular requests (line 9-10). It is important to note that the cutoff on popular requests is reached when the cap on aggregate cache capacity is touched, controlled by MAX_ALLOC_DIVISOR factor (line 9); bigger values lead to fewer assignments while a value of 1 uses the whole cache for assignment.

Initialization

```
1   numServers       = 3       //number of servers
2   currentServer    = 0       //server chosen for policy
3   reqIdx           = 0       //Current Request index

4   CacheSize        = 5000    //from configuration file

5   totalObjAlllocated = 0
6   objectAlloc[] = [0]        //0 objects allocated initially on each server

7   maxObjectAlloc[] = [CacheSize]    //maximum objects allowed per server

8   MAX_ALLOC_DIVISOR = 3     // cap total capacity by dividing this factor
```

Figure 4.2: Basic distribution algorithm initialization

Algorithm

```
9    while totalObjAllocated < totalCapacity/MAX_ALLOC_DIVISOR and
10         reqIdx < # of httpRequest
11
12   do
13
14     //pick the next most frequently access request
15     curReq = httpRequest[reqInd]
16
17     //get corresponding objects for this req
18     objects = reqToResMap[curReq]
19
20     //check if the current server has the capacity to cache all objects
21     if objAllocated[currentServer] + objects < maxObjAlloc[currentServer] then
22
23            //create loadbalancer rule for this request
24            LBPolicy [curReq] = {currentServer}
25
26            //create a cache policy for each object with LONG_TTL
27
28            foreach cacheKey in objects do
29
30                  currentServerASPolicy [cacheKey] = LONG_TTL
31
32            end //for
33
34            // update counters
35            objAllocated[currentServer] += # of new objects assigned
36
37            totalObjAllocated += # of new objects assigned
38
39            //move to the next server
40            currentServer = (currentServer+1) % numServers
41
42            //move to next popular request
43            reqIdx++
44
45     else
46
47            //choose another server as currentServer has max-out
48            currentServer = (currentServer+1) % numServers;
49
50     end
51
52   end //main loop ends
```

Figure 4.3: Basic distribution algorithm

4.4 Replication Strategy

Our previous algorithm will work well if the processing power of the application servers is not bottlenecked. However, it may not perform well if some popular requests are directed to one particular server, which eventually gets saturated. This could adversely affect system throughput as one server might be overloaded while others remain underutilized. To overcome this problem we augment our basic distribution algorithm with replication, where the most popular requests are assigned to multiple servers. We term this algorithm *Replication Strategy*. Selecting the most frequently accessed requests and assigning them across the cluster balances the load on application servers under peak load. The load balancer policy generated for popular requests contains multiple servers instead of one, allowing the load balancer to evenly distribute such requests.

Finding the appropriate request set for replication is very important, because of the limited cache capacity. We cannot simply replicate all popular requests until the cache is filled. As mentioned in the pure distributed strategy, we reserve only a portion of the cache for policies while keeping the rest of the cache available to other requests. Now, we provision some cache for replication leaving the remaining cache till cut-off for distribution. The Figure 4.4 shows how our virtual cache capacity is partitioned per server.

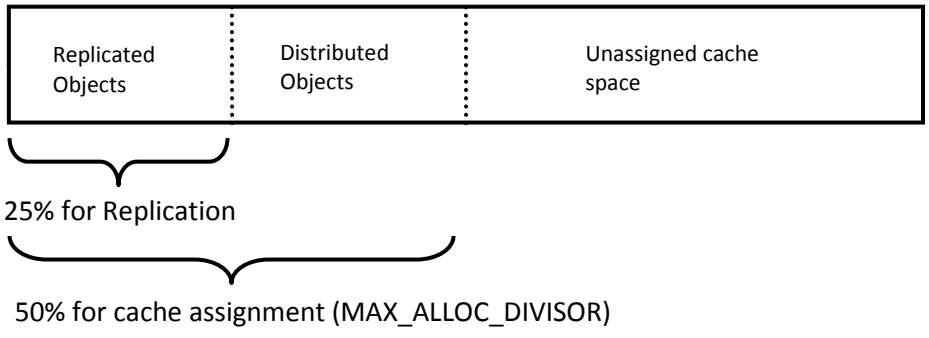


Figure 4.4: Cache partitioning for policy assignments

The cache space provisioned for replication is configurable. However, as the workload might vary, choosing a fixed value beforehand is not the best option. Therefore, we adjust the replication factor dynamically. Below we outline the important steps to compute an effective replication percentage.

- **DesiredReplication (DR)** is the percentage of cache reserved for replication, configured at startup. For example, if 10,000 objects were accessed during the last interval and this factor is 20%, then objects (associated with popular requests) are replicated in the cluster until 20% of the capacity is filled. This value is oblivious of workload and needs adjustments.
- **UniqueResourcesCount** is the number of unique cache object accessed during the last interval.
- **ObjectsPerSlot (OPS)** is calculated as the ratio of UniqueResourcesCount to TotalCacheCapacity. This factor is used to adjust replication. If OPS is less than 1 then the entire resource set (accessed objects) is cacheable, if the factor is greater than 1 than we have more objects and less capacity. If OPS is 1/numServers then all objects could be replicated across cluster.
- **EffectiveReplication (ER)** is the normalized replication and is calculated as

$$ER = DR / OPS$$

For example if the desired replication is 20% but OPS is 2 then our effective replication reduces to 10%. But if OPS is ½ then effective replication jumps up to 40% (or the limit set for the caching policy). Hence, this heuristic tweaks replication factor so that under random workload (resulting in more objects accessed) replication is reduced. On contrary, if only popular requests are made, resulting in fewer objects than available cache space, replication is increased to achieve uniform load distribution.

Figures 4.4 and 4.5 show the algorithm in action. This strategy starts in replication mode (line 9), where one request is assigned to all servers. During the mode,

replication is achieved by only increasing request index (reqIdx) once a request is assigned to all servers (line 57-59). If the cluster size is big then replication could be reduced to k servers with a minor change in the algorithm. Objects assigned during replication mode are given STABLE_TTL (line 33-38), so they tend to outlive other cached objects stored with smaller TTL. The effective replication (ER) factor is computed during initialization (line 11-13). Upon reaching the effective replication threshold (line 51 - 54) the strategy switches to distribution mode (line 56). From here on, the algorithm works just like our basic distribution strategy. This algorithm (Figures 4.5 and 4.6) with the replication ad-on is shown below. The enhancements are highlighted.

Initialization

```
1   numServers       = 3       //number of servers
2   currentServer    = 0       //server chosen for policy
3   reqIdx           = 0       //Current Request index

4   CacheSize        = 5000    //from configuration file

5   totalObjAlllocated = 0
6   objectAlloc[] = [0]        //0 objects allocated initially on each server

7   maxObjectAlloc[] = [CacheSize]    //maximum objects allowed per server

8   MAX_ALLOC_DIVISOR = 3     // or whatever configured

9   repMode    = true         //replication mode or distribution mode

10  uniqueObjects = getUniqueObjects()  //unique object count

11  OPS= uniqueObjects/totalCapacity     // objects per slot

12  DR         = .1          // desired replication (configurable)

13  ER         = DR/OPS      // effective replication
```

Figure 4.5: Replication strategy initialization

Algorithm

```
14   while totalObjAllocated < totalCapacity/MAX_ALLOC_DIVISOR and
15         reqIdx < # of httpRequest
16
17   do
18     //pick the next most frequently access request
19     curReq = httpRequest[reqInd]
20
21     //get corresponding objects for this req
22     objects = reqToResMap[curReq]
23
24     //check if the current server has the capacity to cache all objects
25     if objAllocated[currentServer] + objects < maxObjAlloc[currentServer] then
26
27            //create loadbalancer rule for this request
28            LBPolicy [curReq] = {currentServer}
29
30            //create a cache policy for each object
31            foreach cacheKey in objects do
32
33                   // Stable TTL if in replication mode else long TTL
34                   if repMode then
35                          currentServerASPolicy [cacheKey] = STABLE_TTL
36                   else
37                          currentServerASPolicy [cacheKey] = LONG_TTL
38                   end
39            end
40
41            // update counters and move to next server
42            objAllocated[currentServer] += # of new objects assigned
43
44            totalObjAllocated += # of new objects assigned
45
46            currentServer = (currentServer+1) % numServers
47
48            //update reqIdx based on repMode
49            if repMode then
50                   //current replication factor
51                   repNow = totalObjAllocated/totalCapacity
52
53                   //if we have reached effective replication switch mode
54                   if repNow >= ER then
55                          repMode = false
56                          reqIdx++;
57                   else if replicated on all servers then
58                          reqIdx++    //move to next popular request
59                   end
60
61            else  //not replication mode
62                   reqIdx++  //move to next popular request
63            end
64     else
65            //choose another server as currentServer has max-out
66            currentServer = (currentServer+1) % numServers;
67
68     end
69
70   end //main loop ends
```

Figure 4.6: Replication Strategy

4.5 Compact Assignment Strategy

In the previous two strategies that we have discussed requests are assigned in a round robin fashion to our application server cluster. This round-robin placement of requests works really well since requests are sorted on frequency resulting in even load distribution. Each server in this scheme gets a fairly equal proportion of total request from the load balancer. However, since several requests could access an object, the object is unintentionally replicated to different servers. For example, if request R1 accesses objects {a,b} and is assigned to application server A1, and request R2 accesses objects {a,c,d} and is assigned to application server A2, then we have it redundantly replicated to two servers with the round-robin server selection.

To overcome the above unintended redundancy, we propose a novel method called *Compact Assignment Strategy* where a request is assigned to a server that contains the most objects for the new request. Therefore, in the above example, R2 and the objects {c,d} are assigned to server A1. Hence, for the above case we save 1 unit of cache capacity. This compaction is beneficial if the cache capacity is small and several objects are accessed by many popular requests. Not employing this strategy for such situation would spread the same objects to different servers, thereby considerably reducing aggregate cache capacity.

Now instead of round robin, the server selection for a request *R1* accessing objects in Set *O* follows the following general rules:

- Find the number of overlapping objects each server has with O, and arrange servers in descending order (i.e. server with the most overlap first).
- Pick the top server and check if it can accommodate non-overlapping objects. If not, pick the next one in the list. This minimizes cache redundancy while at the

same time ensures that we don't run over the cutoff set for allocation per cache.

- If two or more servers have the same number of overlapping objects, pick the one with the least number of assigned objects to break tie. This contributes to more even assignment and prohibits a situation where one server is constantly assigned all the requests (and objects).
- If no overlap is found then choose the one with the least number of assigned requests. This follows the same reasoning as the previous rule, but instead we choose request counter as a measure to even out the distribution.

This server selection scheme is only used in the distribution phase of assignment. For the replication phase, we don't care about compactness as the intent is redundancy, and simple round robin suffices.

Figure 4.6 shows the enhanced algorithm with compact assignment strategy. The initialization is pretty much the same except now we maintain counters of request assigned per server (line 79). If we are in non-replication mode then we calculates the object overlap for each server (line 17-21) and choose the candidate server with the most overlap (line 24-28). We also tag objects (line 20) that are already assigned so that we do not assign them again (line 71-73). During this step we also discard the servers with not enough capacity for new objects. There is a high chance that more than two servers have the same overlap because of the replication phase that precedes distribution phase. We break such ties by selecting a candidate with fewer overall objects (line 30-37). If there is no overlap then we simply choose the one with least assigned request (line 44-53). The rest of the algorithm generates policies for the selected server.

Algorithm

```
1  while totalObjAllocated < totalCapacity/MAX_ALLOC_DIVISOR and
2       reqIdx < # of httpRequest do
3
4       //next most popular request and its objects
5       curReq = httpRequest[reqInd]
6       objects = reqToResMap[curReq]
7
8       candidateFound = false
9
10      //only used in Non-Rep Mode
11      if not in repMode then
12           matches[] = [0]          //initialize matches counter to 0
13           candidateServer = undef //candidate Server index
14           max = -1                 //max overlap so far
15
16           foreach server in servers do
17                 //check for overlap and tag matched objects
18                 foreach cacheKey in objects do
19                       if cacheKey found in server then
20                             alreadyAssigned.put(cacheKey)
21                             matches[server]++
22                       end
23                 end
24
25                 //check if the server has enough space left
26                 if matches[server] > max and objAllocated[server] +
27                       objects - matches[server]) < maxObjAlloc[i] then
28
29                       max = matches[server]
30                       candidateServer = server
31
32                 else if matches[i] = max then
33                       // break ties by choosing least assigned objects
34                       if cacheObjAllocated[server] <
35                            cacheObjAllocated[candidateServer] then
36
37                                 candidateServer = server
38                       end
39                 end
40           end
41
42           if candidateServer is not undef then
43                 candidateFound = true
44                 currentServer = candidateServer
45           else
46                 //find the least request-allocated server
47                 candidateServer = 0
48                 foreach server in servers do
49                       if reqAllocated[server] <
50                        reqAllocated[candidateServer] then
51
52                             candidateServer = server
53                       end
54                 end
55                 currentServer = candidateServer
56           end
57      end
```

Continues…

```
58  //check if the current server has the capacity to cache all objects
59    if candidateFound or objAllocated[currentServer] + objects <
60                          maxObjAlloc[currentServer] then
61
62          //create loadbalancer rule for this request
63          LBPolicy [curReq] = {currentServer}
64
65          //create a cache policy for each object
66          foreach cacheKey in objects do
67
68                 // Stable TTL if in replication mode else long TTL
69                 if repMode then
70                        currentServerASPolicy [cacheKey] = STABLE_TTL
71                 else if cacheKey not in alreadyAssigned then
72                        currentServerASPolicy [cacheKey] = LONG_TTL
73                 end
74          end
75
76          // update counters and move to next server
77          objAllocated[currentServer] += # of objects assigned
78
79          totalObjAllocated += # of objects assigned
80
81          reqAllocated[currentServer]++
82
83          currentServer = (currentServer+1) % numServers
84
85          //update reqIdx based on repMode
86          if repMode then
87                 //current replication factor
88                 repNow = totalObjAllocated/totalCapacity
89
90                 //if we have reached effective replication switch mode
91                 if repNow >= ER then
92                        repMode = false
93                        reqIdx++;
94                 else if replicated on all servers then
95                        reqIdx++    //move to next popular request
96                 end
97
98          else  //not replication mode
99                 reqIdx++  //move to next popular request
100         end
101    else
102         //choose another server as currentServer has max-out
103         currentServer = (currentServer+1) % numServers;
104
105    end
106
107 end //main loop ends
```

Figure 4.7: Compact assignment strategy

4.6 Supplementary Load Balancer Assignment

So far, our assignment algorithms stop when the aggregate cache assignment reaches a given threshold. We do so to allow other types of request their fair share of cache, and to handle the unpredictability of the workload. In case of the replication strategy, the reserved cache space fills up much faster because of redundancy. This can be problematic as we are not accommodating other popular requests in our policies which may be queried frequently in the future. The load balancer treats any other request, for which there exist no rule, as an unpopular request and distributes it randomly or in round-robin fashion to all application servers. Subsequent requests, whenever forwarded to servers other than the original one (where corresponding objects were cached) are served much slowly because of remote fetching, which increases response time and creates unnecessary overhead on application servers.

To remedy this situation we generate load balancer policies for unassigned but frequent requests, with no corresponding cache policies. The load balancer will now send all such requests to a particular server, thus avoiding remote calls. This supplemental strategy (Figure 5.6) could be added to all our algorithms.

We introduce a new configurable variable LBUrlFreq (line 1) that specifies the minimum request frequency for which we generate additional load balancer policies. Setting its value to -1 disables this phase. The unassigned requests in our popular list having at least the LBUrlFreq frequency are assigned to application servers (line 12-15). The least request-allocated server is chosen (line 9) to balance out any unevenness in previous phases. For non-compact strategies we simply stick to round-robin selection.

Algorithm

```
1    if LbUrlFreq != -1 then
2
3       while reqIdx < # of httpRequest do
4
5             //pick current object
6             curReq = httpRequest[reqInd]
7
8             //chose the least request-allocated server
9             currentServer = getLeastAllocatedServer()
10
11            //only allocate if request frequency is more than LbUrlFreq
12            if frequency of curReq >= LbUrlFreq then
13
14                  //create loadbalancer rule for this request
15                  LBPolicy [curReq] = currentServer
16
17                  //update request allocated counter
18                  reqAllocated[currentServer]++
19
20                  //move to the next request
21                  reqIdx++
22
23            else
24                  break;
25            end
26
27      end //while
28
29   end
```

Figure 4.8: Load Balancer-Only assignment

4.7 Complexity

The complexity of our algorithms is linear and all algorithms run in O(n) where n is the number of popular requests. In the first two strategies, the Basic Distribution and the Replication Strategy, the server selection is round robin which takes constant time. Also, any lookup in algorithm is from a hash map which is constant. The Compact Assignment Strategy, however, is a tad slower as it has to find the overlap. For S

servers and O objects in the current request, this requires S * O hash lookups or O(S*N). As the number of objects accessed in any given request is usually not large, and the number of application servers is reasonably small, we contend that the selection of a candidate server is of acceptable speed. It is also worth noting that we don't exhaust our full list of popular request, but only a limited subset.

4.8 Simplified Policy Setup

In some cases we may want to reduce complexity and overhead involved in managing policies at the application server. There are several scenarios where we can totally eliminate policy generation for the application server:

- Replication is not required. The basic distribution algorithm distributes requests in a sticky manner, maximizing local cache hits.

- There is no need for differentiated TTL values. Without application server policies all objects are stored with SHORT_TTL with equal chance of eviction.

- A different caching implementation is used, which is totally oblivious of policies and thus a mere simplified load balancer-only strategy suffices.

- The request distribution policies don't really need a complimentary distributed cache environment. Even in non-distributed caching environment this would yield benefits similar to cookie based sticky connection.

Also note that if replication is needed, we can't have an oblivious application server as our default behavior would not replicate objects retrieved from remote cache. Just like above, we can have a setup where we only generate caching policies without caring for request distribution. Possible scenarios could be

- Popular requests are replicated to all servers. Sending such requests to any server would be served totally by the local server. Here, we keep the replication threshold high enough so that most popular requests are replicated leaving no space for the distribution.

- The application servers are running in a virtual environment hosted on the same physical machine with close to none communication latency. The elastic server setup in Amazon EC2 [25] is such a configuration. In this case remote fetch would be nearly as fast as local fetch except for the indirection cost. We can generate caching policies that would make full use of the aggregate cache capacity.

Chapter 5

Experimental Results

5.1 Introduction

This chapter evaluates the request-centric strategies discussed in the last chapter. We first describe the hardware and software environment used for the experiments. Then we give a brief overview of the benchmark we used and the implementation details of our infrastructure. After that we show and compare performance results obtained from different strategies and test configurations.

5.2 Experimental Test Bed

We conducted our experiments on a cluster of 9 nodes. We used a dedicated machine for each of our servers, namely, the load balancer, the application servers (from 3 to 5 instances), the database server, the analysis server, and the client. The hardware and OS specification of all the nodes in our cluster is identical and is shown in Figure 5.1

CPU	Intel(R) Core(TM)2 CPU 6700 @ 2.66GHz
RAM	8 GB 667MHz memory
Network card	Intel 82540EM Gigabit Ethernet Controller
Network switch	3Com Baseline Switch 2948 SFP 1Gbps
Operating System	Fedora Linux Distribution

All the nodes are diskless and access the same disk (Seagate Cheetah Ultra320 15K RPM Hot-Plug Drive) over LAN. We have used JBoss 5.1 as our application server. Each instance of the JBoss application server runs on its own separate node but accesses the same disk (where instances are installed). The application servers run on the 64bit version of Java Development Kit 1.6 (JDK 1.6) and Java Enterprise Edition 1.5 (Java EE 5). We have used PostgresSQL 8.4.1 (64 bit) as our database server.

5.2.1 Implementation Details

The load balancer is behind the application servers and is the only point of access to the application for the clients. We have used Apache 2.0 web server with the load balancing module. The default load balancing algorithm is weighted round-robin. Since the serving capacity of each server is identical we have given each server the same weight (i.e. 1:1:1 for 3 servers). When the load balancer server receives the policy it uses the request counting algorithm [26], which works in the following way

- For each application server an *lbstatus* variable is kept, initialized to 0, for request counting. The lbstatus of the chosen target server is decreased by number of servers – 1, while lbstatus of all other servers is increased by 1.
- If there is a rule for the incoming request the load balancer sends the request to the target server.

- If there is no rule for the incoming request, the candidate target server is chosen with the highest lbstatus. This ensures that the server least requested is chosen for an unassigned request.

The analysis server runs on a dedicated machine, configured to receive remote logs from application servers. We have used the remote logging capability of the Log4J logging library [27], which is very efficient. Log4J creates a persistent TCP connection between each application server and the analysis server for remote logging.

5.2.2 Benchmarking Suite

To evaluate the performance of our algorithms we have used a RUBiS benchmarking suite [28]. RUBiS is a prototype implementation of an auction site modeled after Ebay (ebay.com), and is primarily used to measure application server performance and scalability. This benchmark is comprised of the following two main components:

RUBiS Application

RUBiS comes with a web application built on the Java Servlet API for request processing. This application is interactive and just provides basic auction site functionality without cluttering the UI. Hibernate is used as data access layer, and is plugged with our enhanced version of EhCache (with distribution and replication support).

Client Emulator

To test the RUBiS application a benchmarking tool is provided, called Client Emulator. It is a workload generation tool that emulates several clients' sessions running concurrently. We are only concerned with read-only requests, thus we use the

browsing mix workload for our experiments. In this workload a typical client session starts with logging into the site, and then browsing different categories, regions, items, bids, comments, and user profiles before logging out. The client emulator parses the response it receives for a client request, finds all links that it can follow to emulate the next user interaction. The emulated client waits for on average 7 seconds before issuing another request. This wait time is called *Think Time* and it is kept to mimic a human interaction on a website.

5.3 Methodology

We have done several experiments to analyze the performance of our strategies against the simple cooperative cache and the standalone EhCache models. All our experiments were run for about an hour using the client emulator for different client and cache configurations. To get reasonably good approximation of performance statistics we only present the results obtained during the stable runtime, which excludes the initial 30% of time spent on cache warm-up and the last 20% of time on cool down. The workload for our experiment only consists of read-only requests, and hence the database remains unchanged throughout a run and for subsequent experiments. The size of our database is about 1 GB while the default size of each individual application level cache is 60MB.

For fair comparison among different caching systems and our strategies we have kept the cache and client configuration the same. Our strategies are implemented on top of cooperative cache. If the application server is running without any cache policy, it behaves just like the cooperative cache and does not replicate anything. Moreover, since we have used a singleton cache instance for all entity types in the cooperative cache (and for strategy runs), we have modified the EhCache to use singleton cache, too.

5.4 Motivation

The purpose of our experiments is to confirm our theoretical reasoning for strategies. As we have discussed in the last chapters, the cooperative cache reduces backend database calls but does not yield the optimal response time due to remote cache hits. Before we move to runtime results it is important to know the factors that affect and differentiate our algorithms from other basic approaches. First of all, we put emphasis on cache hit-rate, and in particular the local cache hit-rate. Figure 5.1 illustrates this point with the latency we obtained for a typical local and remote fetch operation.

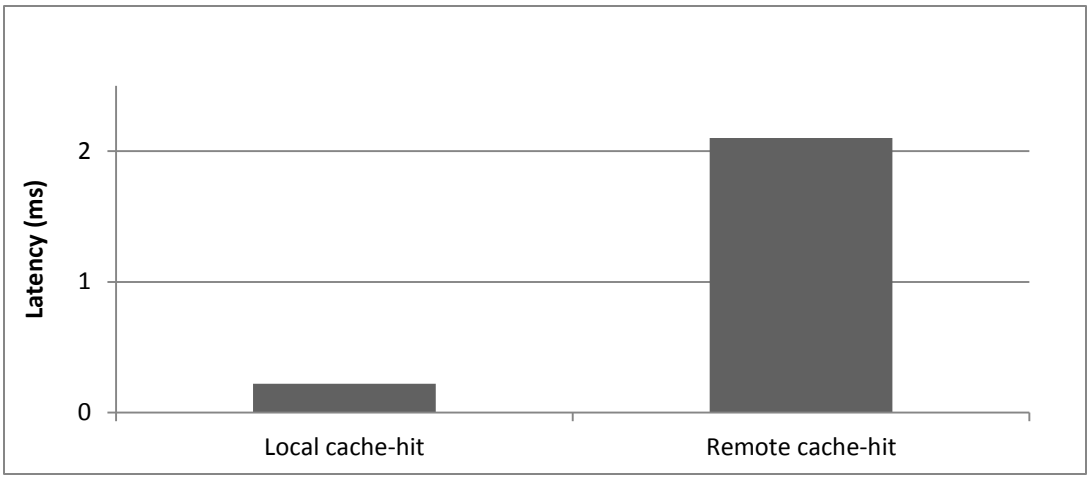


Figure 5.1: Comparison of local and remote cache access latency

The difference between a local cache access and a remote one is quite large. In fact, we can access about 8 objects from the local cache for the time it takes to access one object from a remote cache. Thus, we have put emphasis on local cache access. It is worth mentioning that under heavy network usage, the remote call latency would go up as it is dependent on network IO, while local cache access latency would remain pretty much constant. Similarly, the database access latency varies much like this. Under minimal load on the database server, the database latency is comparable to

that of a remote cache access. Under heavy load it is much higher. If the database saturates, the response time could go up to tens of seconds.

5.5 Experimental Results

Our first experiment demonstrates the advantage of a clustered cache against standalone cache. Figure 5.2 primarily illustrates how the cooperative cache performs against EhCache on a cluster of 3 nodes with round-robin load balancing for 600 and 900 concurrent clients.

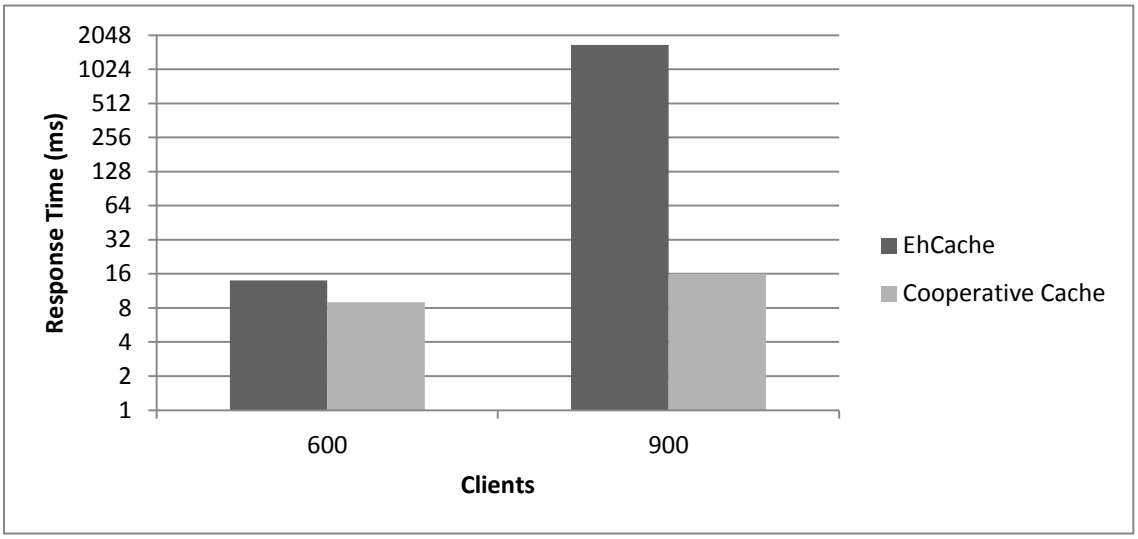


Figure 5.2: Average response time for different caching schemes

It is evident from the figure that clustering application server caches does help greatly and gives the application an immediate performance boost. For smaller workload of 600 clients, both caching schemes result in less than 15ms response time with cooperative cache approximately 36% faster. However, for 900 clients' workload, the average response time for a non-clustered EhCache is over 1.5sec, while it is under 16ms for the basic cooperative cache which is about 100 times faster. This

performance gain is because of a much bigger application level cache (3 times) for the cooperative cache because of cache clustering. The application with a standalone EhCache saturates at this point because of frequent cache misses overloading the database. The saturation is evident from Figure 5.3 which clearly shows lesser throughput for EhCache compared to the cooperative cache for 900 clients.

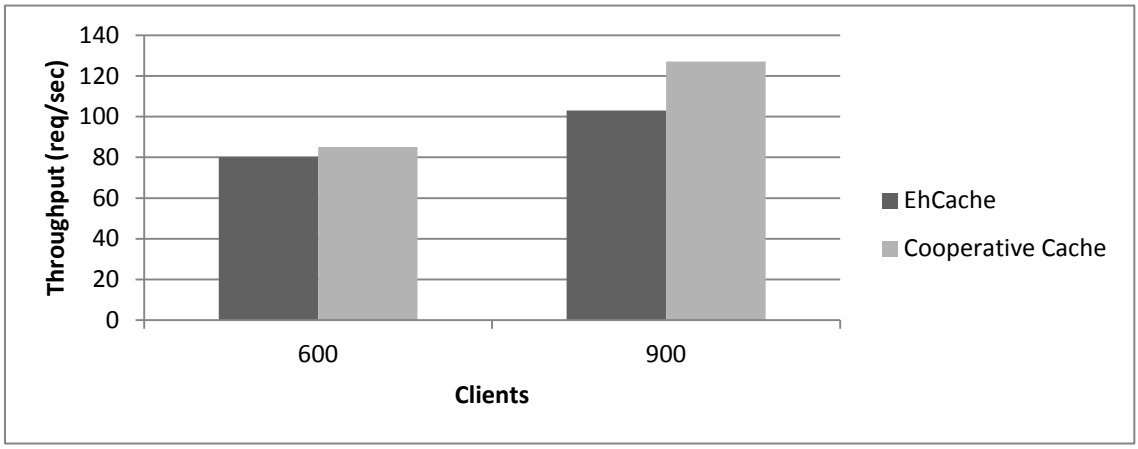


Figure 5.2: Throughput for different caching schemes

5.5.1 General Comparison

We did experiments with different number of clients to see how our strategies perform under various workloads. Figure 5.3 shows the average response time for cooperative cache and our different strategies from 900 to 2000 clients. In this comparison we have the basic distribution strategy with no replication (DistOnly), the compact assignment strategy with 10% replication (CompactDistRep), and the compact assignment strategy with no replication, which essentially makes it compact distribution strategy (CompactDistOnly). In all these strategies, assignment was capped to half of the cache.

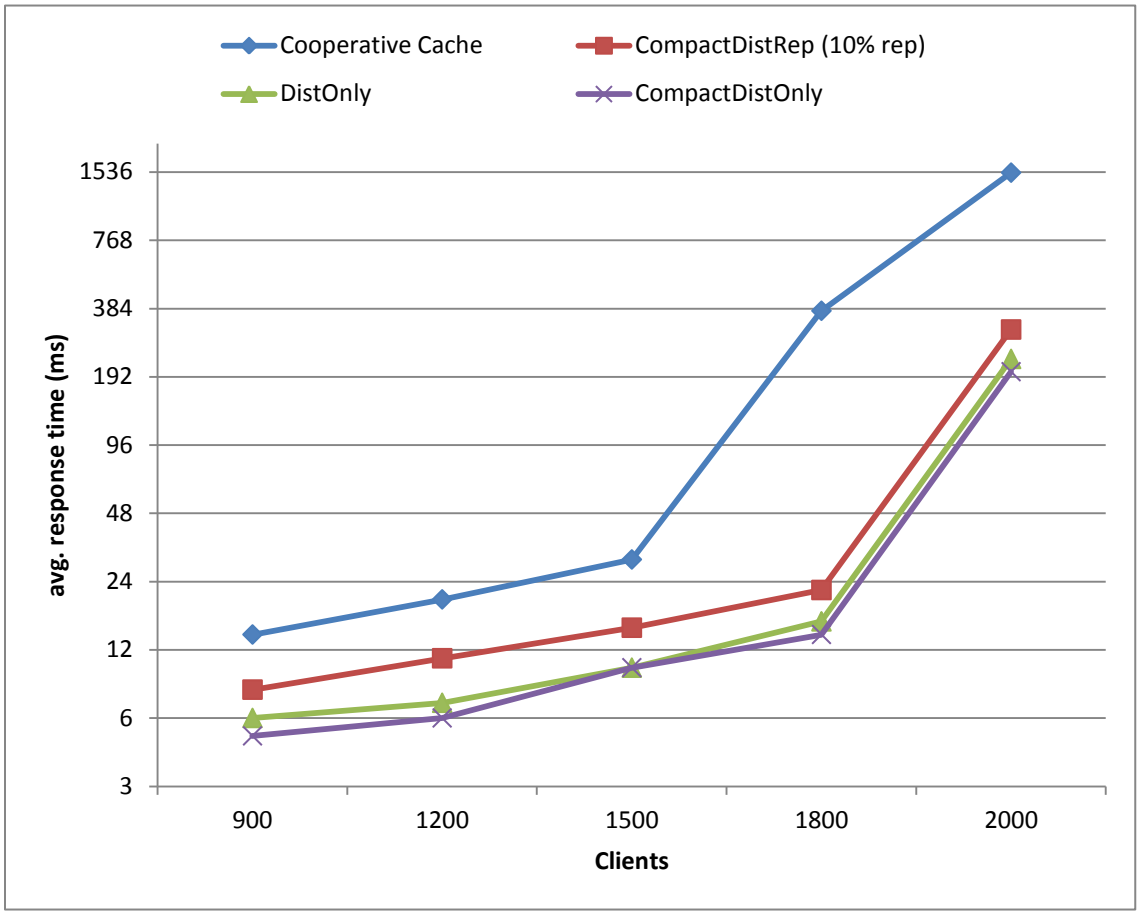


Figure 5.3: Response time of strategies for different clients

The cooperative cache with its round-robin load balancing algorithm performed well under 1500 clients, after which the average response time shot up. For 1800 clients, the cooperative cache reached its highest throughput but couldn't maintain it for 2000 clients where it saturated and response time went over 1.5 seconds. All our three strategies performed better than the cooperative cache for all loads primarily because of our content-aware load distribution. These strategies performed seamlessly well under 1800 clients' load but couldn't maintain the same fast response time for 2000 clients when cache evictions increased resulting in increased database activity.

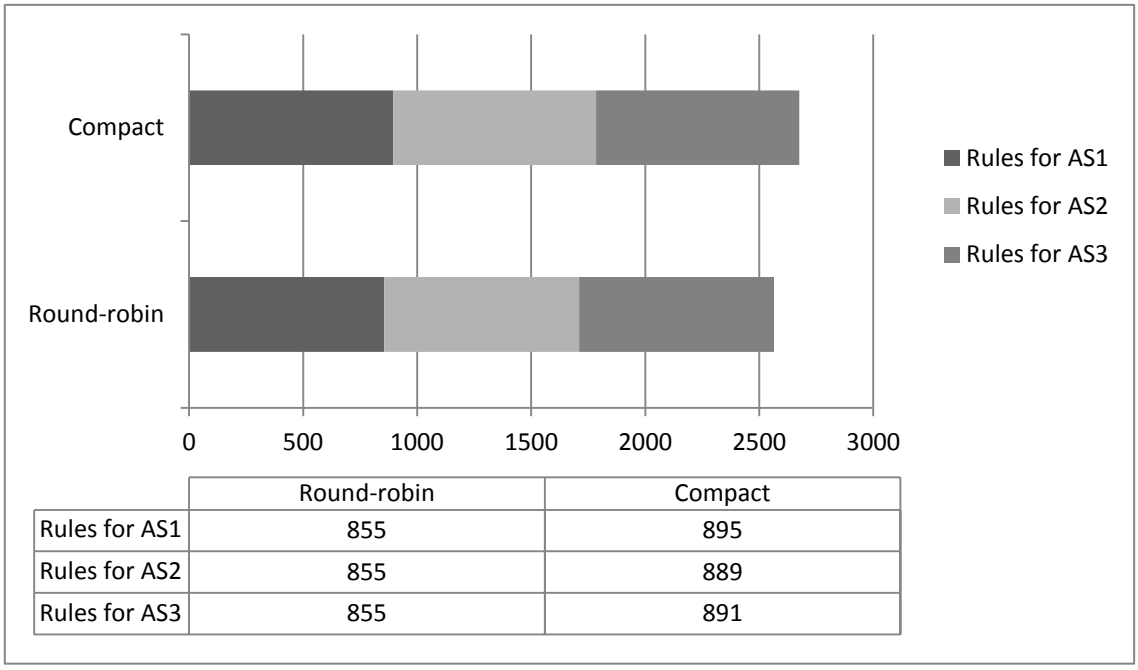
Nonetheless, the response times for all of the strategies remained faster than that of cooperative cache for 1800 because of higher local cache-hits. The CompactDistRep did fairly well but always underperformed compared to the other two strategies. This is because with replication, the total number of popular requests that were being assigned to application servers was less than non-replication strategies. Consequently, when replication strategy came close to saturation its response time was about 30% slower than distribution strategies.

The DistOnly strategy and the CompactDistOnly strategy performed equally well for smaller number of clients, with average response time less than 16 milliseconds till 1800 clients. For 2000 clients load, the average response time jumped for both these strategies. In this case, the CompactDistOnly strategy performed 10% better than DistOnly strategy because the load balancer was able to send requests to a server with a greater chance of local cache hits. However, we couldn't see the anticipated improvement from compact assignment because of the simplistic nature of the RUBiS web application where not a lot of requests access common objects. But we still do contend that with applications where there is a greater overlap of resources in requests the compact assignment would be able to localize requests better.

5.5.2 Compact vs. Round-Robin Assignment

Next we did an experiment to see the benefits of the compact assignment over a naïve round-robin assignment during policy generation (see Section 4.5 and 4.6 of Chapter 4). Our intent here is to show compactness by way of number of policies generated with the same configuration. Figure 5.3 shows the number of load balancer rules generated for each server with round-robin and compact assignment. With round-robin assignment we reach the cache cap faster than compact assignment and the algorithm stops further policy generation. This is because the round-robin assignment

overlooks already assigned objects and inadvertently replicates them to other servers, while compact assignment avoids this redundancy resulting in the generation of 4.2% more rules.



	Round-robin	Compact
Rules for AS1	855	895
Rules for AS2	855	889
Rules for AS3	855	891

Figure 5.4: Round-Robin vs. Compact Assignment Strategy

Another insightful result of the policy distribution in Figure 5.4 data table is the uniformity of compact assignment. The number of rules generated for each target server is very close to each other with the standard deviation of 3.05 rules. This asserts that compact assignment does a fairly good job at generating even yet compact distribution. On contrary, the distribution from round-robin is ought to be even because of its very nature. For 1800 clients, the average response time for compact and round-robin assignment strategy was 21 and 24 milliseconds, respectively. Clearly, the compact assignment strategy performed better, with about 12.5% faster response time than the round-robin assignment. We can attribute this improvement to the 4.2%

additional rules that compact assignment was able to localize to specific servers resulting in more local cache-hits.

5.5.3 Cache Capping for Assignment

To measure the behavior of capping the cache for assignment we performed another experiment by varying the cap. This way, the assignment continued longer and generated more policies. The Figure 5.5 illustrates the effect of a larger cap on response time when benchmarked for 2000 clients.

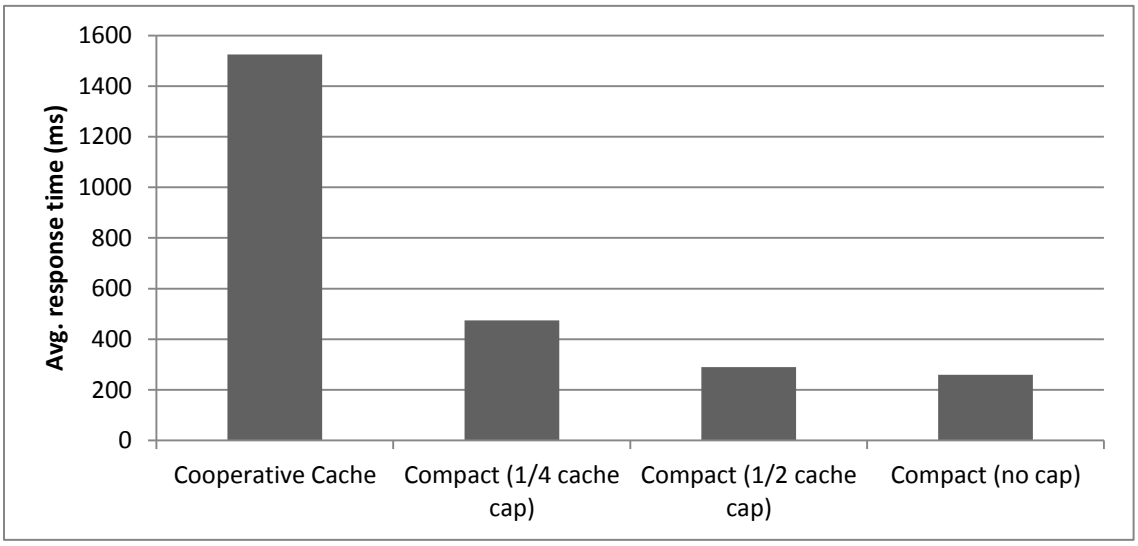


Figure 5.5: Avg. response time variation because of cache capping

The compact assignment algorithm performed better when the cap was changed from a quarter of the cache to half of the cache, by approximately 40%. This is mainly because raising the cap generated twice as many policies per server which helped the load balancer to localize more request to specific servers. Without any cap, we saw a further improvement of 11% over half cache cap. The response time for the simple cooperative cache was much higher, and is shown here to compare the effect of

content aware load balancing against round-robin one. Figure 5.6 shows the proportion of local to remote cache hits for each scenario.

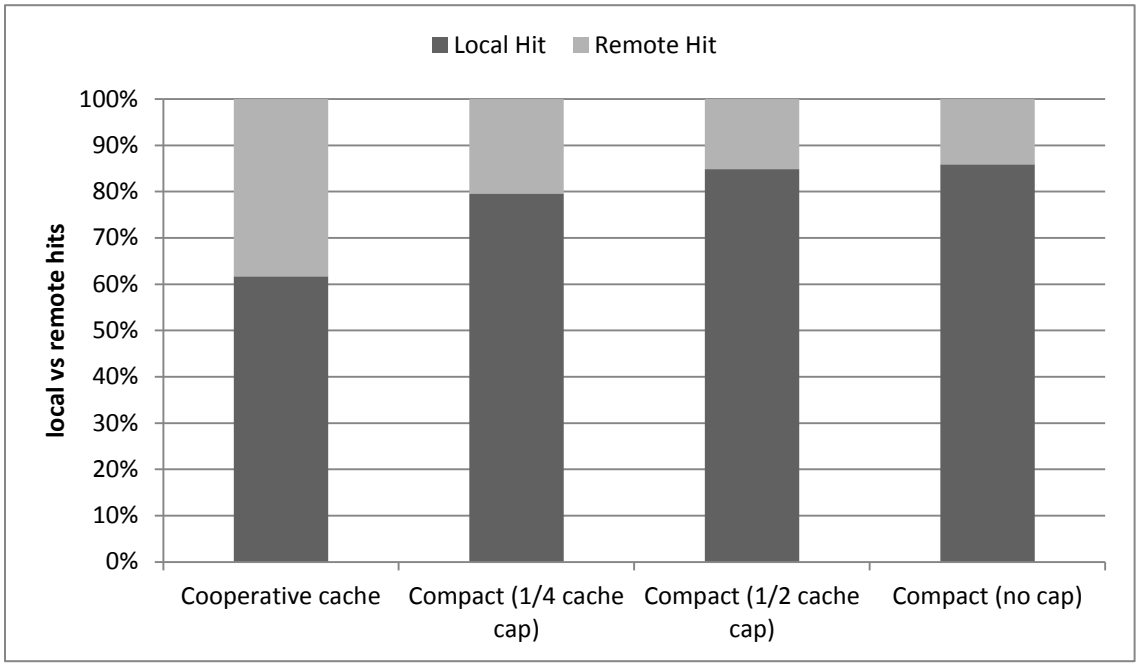


Figure 5.6: Ratio of local to remote cache hits

In the cooperative cache, we got 61% local cache-hit which is not as good as any of our strategies. When the assignment was restricted to $1/4^{th}$ of the cache the local cache hit-rate improved to about 79%. We were able to squeeze out more local cache-hits when the cap was doubled. In this case, local cache hit-rate got up to 85%. When we removed the cap altogether the local cache-hit rate got up to 86%, a slight increase but good enough to give us an 11% improvement in average response time. From analysis log, we also found out that when the cap was set to half the strategy was able to generate policies for all requests that were accessed 5 times or more thus removing the cap resulted in little improvement.

5.5.4 Scalability

Our next two experiments are geared towards scalability. In the first case, we ran our experiment on a cluster of 3 application servers and then on 5 application servers with identical cache and client configuration. As intended, the addition of 2 more nodes to the application tier gave us smaller response time. Figure 5.5 shows this improvement in average response time of the cooperative cache and the compact strategy for 1800 and 2000 clients.

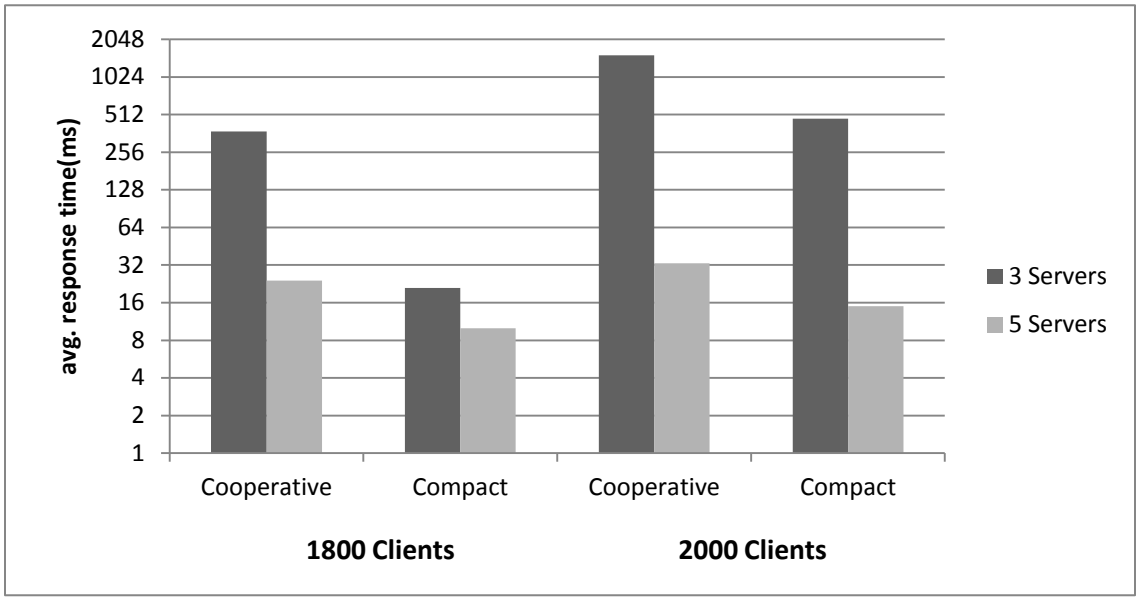


Figure 5.7: Avg. response time for 3 and 5 servers

The response time fell for both the cooperative cache and the compact strategy. The improvement is appreciable in case of cooperative cache, where response time dropped by 93% and 97% for 1800 and 2000 clients, respectively. This plunge in response time is because of the increase in aggregate cache capacity with the addition of 2 nodes resulting in higher hit-rate and fewer evictions. And because of higher hit-ratio the database was accessed less frequently in 5 servers than 3 servers, bumping up the throughput by nearly 8% in case of 2000 clients. We also witnessed good

improvement for our strategy, where response time for 5 servers was 96% faster than that of 3 servers, and about 54% faster than that of cooperative cache for 5 servers, for 2000 clients. The result clearly demonstrates scalability when cluster size was increased and also shows that strategy performed better than cooperative cache because of more local cache hits.

Since the addition of nodes essentially increased the aggregate cache size, we performed another experiment by keeping the cluster size to 3 nodes and increasing the cache size. We chose the basic distribution strategy for this one and ran experiments for 2000 clients with 60MB and 110MB caches and 1/4th cache cap. Figure 5.8 shows the average response time for the mentioned configuration. We observed an improvement of 84% in case of the cooperative cache and about 94% in case of our distribution strategy. We can clearly see that increasing the cache capacity has a direct impact on the response time for both the cases. The cache statistics also showed an increase of 27% in the local cache hit-rate and a decrease of 15% in remote cache hit-rate for our distribution strategy which asserts that most of the request were satisfied from the cache and particularly, the local cache.

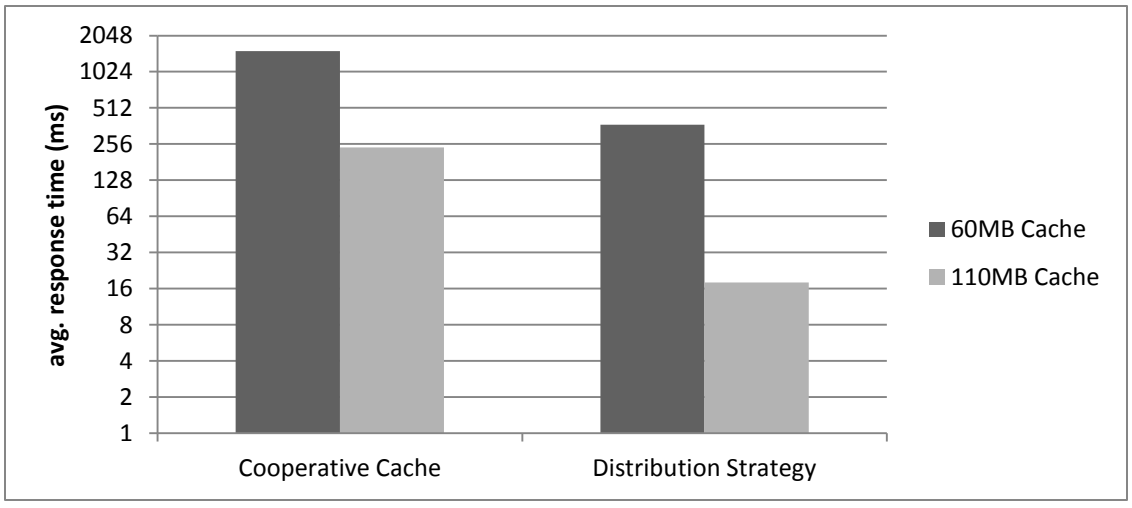


Figure 5.8: Effect of cache capacity on response time

5.5.5 Replication

In Section 5.5.1 we found that Replication strategy didn't perform that well compared to distribution strategies because the application servers never got saturated as RUBiS application has a very simple business logic. Therefore, we modified the RUBiS application to perform some CPU intensive task. For this, we introduced a non functional CPU task in BrowseCategories request to emulate complex business logic. We also modified the workload so that this request was accessed more often making it a popular request suitable for replication. Figure 5.9 shows the improvement when popular requests were replicated, for 2000 clients with the modified workload. It is evident that distributing a popular resource-hungry request to just one server performs worse than replicating it on multiple servers. The total average response time in the case of replication was 3 times faster than that of Distribution-Only strategy.

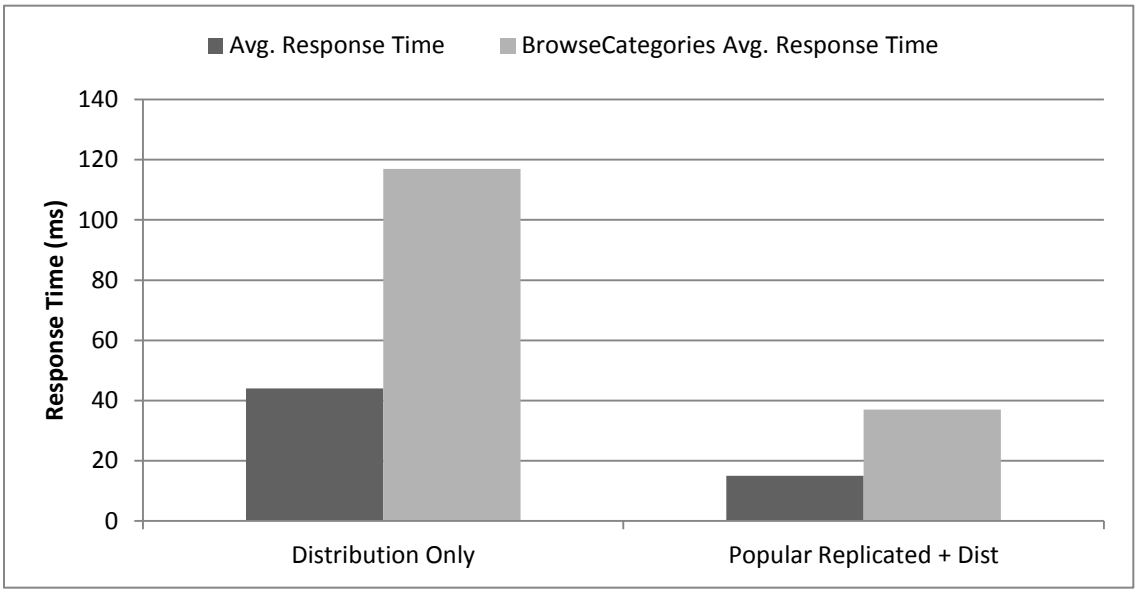


Figure 5.9: Popular replicated strategy against distribution only strategy

In the case of only distribution, the BrowseCategories request got assigned to application server 1 while in the case of replication, it was assigned to all three application servers (with corresponding objects replicated). The Distribution-Only strategy resulted in saturation of application server 1, whose CPU became the bottleneck, shooting up the average response time to 117 milliseconds for the BrowseCategories request. Also in this case, other application servers were underutilized. On contrary, when BrowseCategories requests were served by multiple servers, none of the application servers got saturated resulting in a much faster response time of 37 milliseconds. Hence, we argue that it is much better strategy to replicate resource intensive popular requests to avoid imbalance in cluster performance and utilization.

5.5.6 Total Database Cacheability

Sometimes the database is not big and there is enough aggregate cache capacity to permanently store almost 100% of the database into the cache. For such a scenario, we evaluate the performance of our strategy against the naïve cooperative cache. We chose a smaller database that could be completely cached in approximately 105 MB of aggregate cache (about 35MB per application server). Figure 5.10 shows the average response for cooperative cache and compact strategy for different workloads.

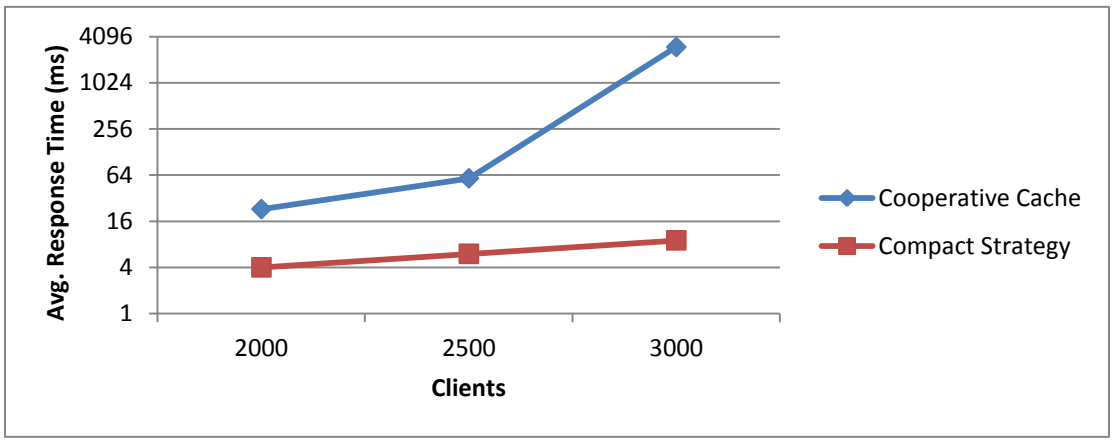


Figure 5.10: Strategy vs. Cooperative Cache on smaller database

After the cache warm up the requests were mostly served from the cache, either local or remote. In case of the cooperative cache, we observed an even hit-rate for local and remote caches, while for the compact strategy the local hit-rate was more than 9 times the remote hit-rate, which explains why the compact strategy always trumped the cooperative cache. For 2000 and 2500 client workloads, the compact strategy resulted in an 82% and 89% faster response time, respectively. Nonetheless, the cooperative cache was able to gracefully handle workload of 2500 concurrent clients with the throughput of about 350 requests per second, nearly equal to that of the compact strategy. The cooperative strategy saturated at 3000 clients while our strategy was still able to handle this workload, which clearly underscores the importance of local cache-hits. Since database access is almost none, the saturation of the cooperative cache for 3000 concurrent clients is attributed to overwhelming increase in remote cache accesses adversely affecting the network latency per remote call. At this point the throughput fell by 15% for the cooperative cache while it jumped to about 427 requests per seconds for our strategy, indicating that content-aware request distribution is essential for achieving high throughputs.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Performance and scalability of multi-tier internet applications have been an active research area as these applications are frequently overwhelmed by an ever increasing user base. Since most of the requests are dynamic in nature and require frequent access to the database, the throughput of a multi-tier application starts to suffer as database becomes the bottleneck. In this thesis, we address these concerns with a holistic caching infrastructure that seeks to improve application performance and enables scalability.

We reduce the bottleneck on the database server by exploiting a distributed cluster-wide application cache, having aggregate capacity much larger than the individual cache, where objects are retrieved from the remote application server cache if they are not found locally. In this thesis we added the support for object replication to this distributed cache, so that popular objects could be accessed from multiple servers

enabling better cluster utilization in case of frequent popular requests. On top of it, we built an application monitoring tool that transparently logs request and cache statistics on a remote analysis server. These logs are processed and analyzed at the analysis server which then generates request distribution policies for the load balancer and caching policies for the application servers. These policies are intended to reduce latency involved in remote cache access by mapping requests to servers in such a way that maximizes local cache-hits. On one hand our monitoring and analysis framework is application-unaware and requires no changes in the application, while on the other it generates content-aware policies for better performance.

In this thesis we focused on request-centric analysis, where we identify popular requests and assign them to specific application server. Our first algorithm, the basic distribution algorithm, assigns popular requests to servers in a round-robin fashion. Next we develop a replication and distribution strategy where the most popular requests are assigned to multiple servers while the others are distributed. Finally, we build a compact assignment strategy where a request is assigned to a server containing the maximum number of associated objects, thereby avoiding unintended object replication occurring with round-robin assignment. All our strategies are extensible and can be easily configured for optimal policies.

We analyzed the benefits of our strategies with the RUBiS benchmark. All our strategies were able to improve application performance under various workloads when compared to the base cooperative cache environment. The application was also able to scale much better under peak workload when policies were applied and yielded a better throughput and response time than the cooperative cache.

6.2 Future Work

The holistic caching infrastructure is very extensible and amenable for a wide range of analysis. For example, one could make use of the request tracing ability of our system to identify user session activity from application access patterns for usability and security analysis.

The analysis could also incorporate other parameters that we log such as average request processing time to filter out requests which are executed faster while prioritizing bottleneck requests during policy generation phase.

In this thesis we have focused on read-only workloads and have not looked at consistency issues involved with object replication in a read-write workload. We believe that consistency mechanisms could be enforced in our caching infrastructure by cache invalidation or update propagation to other caches.

Bibliography

[1] Historic Election Day Sets Traffic Records. [Online]. http://newteevee.com/2008/11/05/historic-election-day-sets-traffic-records/

[2] Ian Evans. (2011) An Introduction to the Java EE Platform. [Online]. http://download.oracle.com/javaee/6/firstcup/doc/

[3] Enterprise JavaBeans Technology. [Online]. http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[4] Ed Ort Rahul Biswas. (2006) The Java Persistence API. [Online]. http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html

[5] Hibernate. [Online]. http://www.hibernate.org/

[6] Neeraj Santosh Tickoo, "Cache aware load balancing for scaling of Multi-Tier Architectures," McGill University, Montreal, QC, MSc. Thesis 2011.

[7] Caching Guide. [Online]. http://httpd.apache.org/docs/2.2/caching.html

[8] Oracle. query result cache in oracle 11g. [Online]. http://www.oracle-developer.net/display.php?id=503

[9] MySQL. The MySQL Query Cache. [Online]. http://dev.mysql.com/doc/refman/5.1/en/query-cache.html

[10] EhCache. [Online]. http://ehcache.org/

[11] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi, "An analytical model for multi-tier internet services and its applications," in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 2005, pp. 291-302.

[12] Bhuvan and Pacifici, Giovanni and Shenoy, Prashant and Spreitzer, Mike and

Tantawi, Asser Urgaonkar, "Analytic modeling of multitier Internet applications," *ACM Trans. Web*, vol. 1, no. 1, May 2007.

[13] Ludmila Cherkasova, Ningfang Mi, Evgenia Smirni Qi Zhang, "A regression-based analytic model for capacity planning of multi-tier applications," *Cluster Computing*, pp. 197-211, 2008.

[14] Xue Liu, Jin Heo, and Lui Sha, "Modeling 3-Tiered Web Services," *Computer Science Research and Tech Reports*, 2005.

[15] Guillaume Pierre, Maarten van Steen, Gustavo Alonso Swaminathan Sivasubramanian, "Analysis of caching and replication strategies for web applications," *IEEE Internet Computing*, pp. 60-66, 2007.

[16] Brad Fitzpatrick, "Distributed caching with memcached," , Seattle, WA, USA, 2004.

[17] Terracotta. Distributed Caching With Terracotta. [Online]. http://ehcache.org/documentation/distributed_caching_with_terracotta.html

[18] Shamir Sultan Ali, "Contquer: An optimized distributed cooperative query caching architecture," McGill University, Montreal, QC, MSc. Thesis 2011.

[19] Vivek S. and Aron, Mohit and Banga, Gaurov and Svendsen, Michael and Druschel, Peter and Zwaenepoel, Willy and Nahum, Erich Pai, "Locality-aware request distribution in cluster-based network servers," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, San Jose, CA, 1998, pp. 205-216.

[20] Sameh and Dropsho, Steven and Zwaenepoel, Willy Elnikety, "Tashkent+: memory-aware load balancing and update filtering in replicated databases," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, Lisbon, Portugal, 2007, pp. 399-412.

[21] Alexander Rasmussen, Emre Kiciman, Benjamin Livshits, and Madanlal Musuvathi, "Improving the responsiveness of internet services with automatic cache placement," in *Proceedings of the 4th ACM European conference on Computer systems*, New York, NY, USA, 2009, pp. 27-32.

[22] Emre and Livshits, Benjamin and Musuvathi, Madanlal Kiciman, "FLUXO: A Simple Service Compiler," in *Proceedings of the 12th conference on Hot topics in operating systems*, Berkeley, CA, USA, 2009, pp. 20-20.

[23] Sanket Joshipura, "Object Based Caching and Load Balancing Strategies for

Multi-Tier Architectures," McGill University, Montreal, QC, MSc. Thesis 2011.

[24] Interceptor Pattern - Wikipedia. [Online]. http://en.wikipedia.org/wiki/Interceptor_pattern

[25] Amazon Elastic Compute Cloud. [Online]. http://aws.amazon.com/ec2/

[26] Apache Module mod_proxy_balancer. [Online]. http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

[27] Log4J Services. [Online]. http://logging.apache.org/log4j/1.2/

[28] RUBiS. [Online]. http://rubis.ow2.org/