INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
Multi-tier Caching of Dynamic Content for Database-driven Web Sites

Karthick Rajamani

Thesis: Doctor of Philosophy
Electrical and Computer Engineering
Rice University, Houston, Texas (October 2001)
Multi-tier Caching of Dynamic Content for Database-driven Web Sites

by

Karthick Rajamani

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Alan Cox, Chair
Associate Professor
Computer Science and Electrical and Computer Engineering

Devika Subramanian
Associate Professor
Computer Science

Peter J. Varman
Associate Professor
Electrical and Computer Engineering

Willy Zwaenepoel
Hasselman Professor of Computer Science
Computer Science and Electrical and Computer Engineering

Houston, Texas
October, 2001
Multi-tier Caching of Dynamic Content for Database-driven Web Sites

Karthick Rajamani

Abstract

Web sites have gradually shifted from delivering just static html pages and images to customized, user-specific content and plethora of online services. The new features and facilities are made possible by dynamic content which is produced at request time. Multi-tiered database-driven web sites form the predominant infrastructure for most structured and scalable approaches to dynamic content delivery. However, even with these scalable approaches, the request-time computation and high resource demands for dynamic content generation result in significantly higher latencies and lower throughputs than for sites with just static content.

This thesis proposes the caching of dynamic content as the solution for improving the performance of web sites with significant amount of dynamic content. This work shows that there is significant locality in the data accesses and computations for content generation which can be exploited by caching to improve performance. This work introduces a novel multi-tier caching architecture that incorporates multiple, independent caching components to enable easy deployment and effective performance over the prevalent multi-tiered database-driven architecture for dynamic content delivery.

The dynamic content infrastructure and the proposed caching strategy is evaluated with e-commerce workloads from the TPC-W benchmark. The evaluation of
the system without caching shows that content generation overheads are dominated by the database component for e-commerce workloads. With multi-tier caching, each caching component overcomes specific overheads during content generation while the combination provides overall improvements in performance significantly greater than the individual contributions. The increased peak throughputs with caching range from 1.58 to 8.72 times the peak throughputs without caching at similar or significantly reduced average response times. At the same load as for the peak throughputs without caching, the response times were reduced by 90% to 97% with caching. The evaluations also establish the effectiveness of the strategy in relation to variation in platform and site configurations. Overall, the proposed multi-tier caching strategy brings about dramatic improvements in performance for dynamic content delivery.
Acknowledgments

I acknowledge with deep gratitude the counseling and support extended to me by my advisor, Prof. Alan Cox, during these many years of my graduate studies here at Rice University. I strongly appreciate his patience and tolerance that afforded me the freedom and flexibility to pursue research in various topics of my interest over the years. I would also like to express my gratitude to the leader of our group, Prof. Willy Zwaenepoel, for his valuable guidance and counsel during this period.

I would like to extend my thanks to my other committee members Prof. Devika Subramanian and Prof. Peter Varman for their support and advise during my dissertation period.

My foray into data mining performance which led to a firm understanding of database systems would not have been possible without the sponsorship and support of Bala Iyer in the IBM Santa Teresa (Silicon Valley) Labs, San Jose, CA. I deeply appreciate his support and counsel through the years.

I would also like to appreciate the support and help extended to me over the years by the Electrical and Computer Engineering Department, the Computer Science Department at Rice, the Office of International Students and Scholars, and all their staff and members.

This work would not have been possible without the support over the years from my many room-mates and friends during the course of my study here at Rice - Ramakrishnan Rajamony, Parthasarathy Ranganathan, Mahesh Kallahalla, Suman Das, Srikrishna Bhashyam, Krishnakiran Mikkavilli, Ravishankar Vadali, Dinesh
Rajan, Brinda Roy, Kota Kishore and Ravi Rao, to mention some. I appreciate their tolerance and genuine friendship that helped me enjoy a home so far away from my family.

I also appreciate the support from my friends at work - Cristiano Amza, Mohit Aron, Charlie Hu, Honghui Lu, Juan Navarro, Johnny Chen and the many juniors whom I seem to have come in closer contact over the last year. I would like to thank, Cristiano Amza, in particular with whom I got to collaborate on many projects.

Last and certainly the most, I would like to thank my family - my father: Raja- mani, my mother: Meenakshi, and my sister: Sukanya - for their unstinting support, faith, and patience during this sometimes testing period. Without their constant encouragement and support I certainly could not have attained my current state in life.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations x
List of Tables xii

1 Introduction 1
1.1 Delivering Dynamic Content - Database-driven Web Sites 1
1.2 Thesis Statement and Contributions 5
1.3 Caching To Improve Web Site Performance 6
1.4 Related Work 10
1.5 Dissertation Overview 12

2 Multi-tier Caching of Dynamic Content 14
2.1 Cache Consistency 16
2.2 Query Cache 18
2.3 Script Cache 21
2.4 DCache - The Complete Response Page Cache 23
  2.4.1 Architecture of Apache 24
  2.4.2 DCache – The Cache Module for Dynamic Content 25
  2.4.3 DCache implementation 27

3 TPC-W - Evaluation Workloads 30
3.1 Cache Characteristics for TPC-W Workloads ........................................ 31
3.2 Implementation .................................................................................. 33

4 Evaluation of Caching in Database-driven web sites .................. 36
  4.1 Experimental Setup .......................................................................... 37
  4.2 Base System Performance ................................................................. 38
  4.3 Caching Performance ........................................................................ 42
    4.3.1 Query Cache .............................................................................. 42
    4.3.2 Script Cache ............................................................................. 47
    4.3.3 Page Cache ............................................................................... 50
    4.3.4 Combining Cache Components .................................................. 54
    4.3.5 Storage Costs for Caching ........................................................... 58
  4.4 Caching on Alternate Platform .......................................................... 59
    4.4.1 Base System Performance ............................................................. 60
    4.4.2 Caching Performance ................................................................. 61
  4.5 Changing Workload Characteristics ................................................. 62
    4.5.1 Database Size .......................................................................... 63
    4.5.2 Number of Items ...................................................................... 66
  4.6 Summary ........................................................................................... 69

5 Conclusions and Future Work ......................................................... 72
  5.1 Summary of Contributions ............................................................... 72
  5.2 Future Work ..................................................................................... 73
    5.2.1 Other Dynamic Content Workloads .......................................... 73
    5.2.2 Cache Implementation with Automatic Dependency Detection .... 74
    5.2.3 Caching Solutions for Server Farms ........................................... 75
B.4  ProductDetail ................................................. 98
B.5  SearchResult .................................................. 98
  B.5.1  By Author .................................................. 98
  B.5.2  By Title ................................................... 99
  B.5.3  By Subject ................................................. 100
B.6  BuyConfirm .................................................... 100
B.7  AdminRequest ................................................. 101
B.8  AdminConfirm ............................................... 101
Illustrations

2.1 Example of the application code change to incorporate QCaché 20

4.1 Performance of the Base System Without any Caching 39
4.2 Response Time Components at Peak Throughput 40
4.3 Performance of Query Cache for Browsing 43
4.4 Performance of Query Cache for Shopping 44
4.5 Performance of Query Cache for Ordering 46
4.6 Performance of Script Cache for Browsing 48
4.7 Performance of Script Cache for Shopping 49
4.8 Performance of Script Cache for Ordering 49
4.9 Performance of Query Cache with Script Cache 50
4.10 Performance of Page Cache for Browsing 52
4.11 Performance of Page Cache for Shopping 53
4.12 Performance of Page Cache for Ordering 53
4.13 Performance of Caching Strategy for Browsing 56
4.14 Performance of Caching Strategy for Shopping 56
4.15 Performance of Caching Strategy for Ordering 58
4.16 Relative Performances of the Two System Configurations without Caching 60
4.17 Relative Performances of Query Cache and Page Cache for the Two
System Configurations ................................................. 62

4.18 Relative Performances of Combined Caching Strategy for the Two
System Configurations .................................................. 63

4.19 Throughput for Different Database Sizes Without Caching ............. 64

4.20 Improvements in Throughput for Different Database Sizes ............... 65

4.21 Breakdown of Response Time for Different Database Sizes .............. 65

4.22 Base System Throughput for Different Number of Items ................. 67

4.23 Improvements in Throughput for Different Number of Items ............ 68
Tables

3.1 Interactions Mix for TPC-W Workloads .................................. 31
3.2 TPC-W Database Tables .................................................. 33

4.1 Load, Throughput, and Response Times at Peak Performance of
   Base System ................................................................. 41
4.2 Query Cache Throughput Comparison At Same or Lower Response
   Times ............................................................................ 45
4.3 Query Cache Statistics at Peak Performance for TPC-W Workloads . 47
4.4 Page Cache Statistics at Peak Performance for TPC-W Workloads . 54
4.5 Number of Concurrent Clients for Peak Performance on the Two
   System Configurations ...................................................... 61
4.6 Configuration for Evaluating Effect of Database Size on Caching . 64
4.7 Number of Concurrent Clients at Peak Performance for Different
   Database Sizes ................................................................ 65
4.8 Configuration for Evaluating Effect of Number of Items on Caching . 67
4.9 Number of Concurrent Clients at Peak Performance for Different
   Number of Items ............................................................. 69
Chapter 1

Introduction

Dynamic content is the driving force behind many new and interesting approaches to publishing information and a wide variety of services on the internet [1, 2]. The ability to utilize powerful programming approaches to generate the content for a response dynamically, i.e., at request time, extends the capabilities of a web site dramatically. It enables the ability to provide user-specific and request-specific content - programs at the site customize the response as desired by identifying or generating content using request time parameters and user-specific options. Request-initiated programs can also perform a variety of tasks on behalf of the user enabling a wide variety of online services such as booking and reservation, e-commerce, trading, banking, messaging, bulletin boards and discussion groups, just to name a few. Overall, the adoption of dynamically-generated content has made web sites more useful, interesting and attractive.

1.1 Delivering Dynamic Content - Database-driven Web Sites

Numerous mechanisms have evolved from the original cgi approach for dynamic content delivery. The cgi approach provided an interface to invoke arbitrary programs to perform request-time computation and content generation. For each incoming request a new process would be created to run the corresponding program and generate the response. Further refinements have reduced process-creation overheads, increased scalability of the web sites, and enabled greater standardization of the
content-generation procedures. Today, database-driven web sites provide the predominant infrastructure for dynamic content delivery. The traditional strengths of the database systems that have brought about this scenario are

- Support for indexed access to large volumes of data.
- Support for reliable storage and retrieval.
- Advanced querying facilities, multiple view management, concurrent accesses to underlying data, and transaction support.

The task of web-site development is simplified and modularized when the source data is available in a highly organized environment as in a database system. Further, increasingly flexible and selective querying facilities of database systems allow the web servers to deliver highly user-specific content (by appropriately querying the database for the relevant data) and enhance the value of the web site to its users or customers. This is critical in the current information-driven economy where the requirement for fast access to just the required information is evenly matched by the need to collect and analyze all possible sources of information.

There are also many situations in which the web servers serve mainly as a front-end to publish content from a database system. For such situations, distributed access to the databases' content is the primary concern and the web servers are just facilitators for data access by enabling distributed database client-server communication through the universal HTTP protocol.

In general, however, database-driven web sites are much more complex than traditional database applications. Database-driven web sites use a three-tiered architecture composed of database servers that support the maintenance and querying
of the data driving the web site, application servers or content generators that handle the creation and provision of dynamic content and services, and web servers that handle the communication with clients across the internet. The demand for dynamically-generated content has caused a tremendous growth in the systems to support it: extensible, modular web servers that can incorporate a variety of modules to implement different kinds of services; application development environments such as PHP, mod_perl, Java servlets (JSP), Active Server Pages (ASP), Enterprise Java Beans (EJB); and extensions to traditional systems, such as databases, to provide interfaces to the new web servers and application development environments. Thus, dynamic content delivery mechanisms are composed of multiple layers of system components which can include a dramatic variety particularly in the content generation layer.

The problem with the technology for dynamically-generated content is that it consumes much more resources, both CPU and memory, than required for static content. When a web page is requested from a site, the server now does not just choose the right set of html and image files to deliver. The server typically performs some combination of the following: computations to identify request-specific content, interactions with other servers on behalf of the request either to fetch some additional data (typically from a database server) or to perform some complicated transaction, and reformatting of the final response data. The common sequence of actions in a database-driven web site is as follows:

- Client requests a particular page, or submits a particular form.

- Web server performs any necessary authentication or permission checks and identifies the file (html, other static files such as image files or program file) that corresponds to the request.
• If the request is for a static component the web server can directly service it. However, if the request requires a program to be executed, the server then proceeds to execute the corresponding program to generate content matching the request. The content generation can happen at the web server itself (often the case with server-side scripting options such as ASP, Perl, PHP etc) or at dedicated application servers (with servlets, EJBs etc). The content generation phase often involves parsing and compilation of the corresponding program matching the request followed by actual program execution with the request-specific arguments.

• If the content generation requires data from the database system, then
  
  – the program sets up (or re-uses) a connection to the database server - might involve network/inter-process communication setup, database authentication, and access privilege verification.
  
  – the program then issues queries to the database system.
  
  – for each query submitted to the database system, the database system parses the query, determines a suitable plan for executing the query (or chooses from previously computed and cached plans), performs the necessary I/O to retrieve the data from disk, performs the necessary query on the retrieved data, commits any updated data to disk if necessary, and finally returns the result to the content generator program.

  – the content generator program then adds any required formatting for the result data.

• Finally, the program prepares the response for the client, while also fetching any additional components from other systems (file system etc).
- The web server then responds to the client with the generated response.

All the increased activity on the servers comes at the cost of the site's performance.

Much has been done to accelerate the performance of static content service - better memory management on servers, using RAID and SAN technologies to address disc performance, using clustering technology to leverage the power of multiple servers, and reverse proxies and content distribution networks to reduce the load on the primary servers. However, little has been done until very recently to address the bottlenecks from dynamic content generation. An enormous increase in resources is required at the servers to achieve the same kind of throughput for dynamic content as can be achieved for static content.

1.2 Thesis Statement and Contributions

I propose that a multi-tier dynamic content caching strategy is the solution to the performance problems with dynamic content delivery. My thesis argues that dynamic content generation overheads can be dramatically reduced by caching and re-using various results produced during the computations for content generation. To take maximum advantage of caches, I propose a multi-tiered caching architecture with different cache components each designed to cache results of a certain kind to alleviate a specific potential source of bottleneck in the content generation process.

My main contributions

- I propose a novel caching strategy that incorporates independent caching solutions in a multi-layer caching architecture. This approach provides significant performance improvement along with the ability to be incorporated with the large variety of content generation methods that are widely in use.
• I provide the first comprehensive analysis of the benefits of dynamic content caching using a complete web-site serving dynamic content. The site is set up for e-commerce with the evaluation performed using workloads from the web-benchmark, TPC-W. I show the value of the different caching components, their synergistic behavior and the effect of the overall strategy combining all the components. I show that we can obtain almost an order of magnitude improvement in the peak performance of the web site using the proposed dynamic content caching strategy.

1.3 Caching To Improve Web Site Performance

Currently, caching is widely employed as an effective solution to improve performance for static content delivery. This takes the form of caching servers at web sites also known as reverse proxies, proxy caches in the internet which also help reduce bandwidth demands and finally the browser caches at the web-site user or client. A more recent advancement in the static caching technologies has been the arrival of content distribution networks (CDN) such as Akamai and Digital Island - technically they can be considered as a network of reverse proxies located at the edges of the internet and shared by multiple web sites. For static content, caching is a straightforward solution both at web sites where the trade-off is predominantly increased in-memory storage to reduce disk reads and at the browser caches where the trade-off is local storage to avoid network latencies. At the proxies many interesting research issues have been addressed over the years including cache management - consistency and replacement policies [3, 4, 5, 6]; prefetching [7, 8, 9, 10, 11] ; and, co-operative caching [12, 13, 14].

Two factors that determine the effectiveness of any caching solution are (1) the
trade-offs between cache storage and management versus benefits of cached accesses, and (2) the extent of re-use of cached results. Given the significant magnitude of network latencies that still exist on the internet, the trade-offs with cache storage and management overheads have been quite favorable for static content caching. For static content, researchers have also shown significant re-use at different levels of the internet cache hierarchy. However, the nature of these two factors is not so clear for dynamic content delivery.

With dynamic content, there is a significant load on the servers just to generate the content. This load results in significantly reduced throughput at the site compared to static content. In addition, loaded servers result in server latencies quite comparable if not greater than network latencies on the internet, further exacerbating the client-perceived latencies. Thus, the role of caches at the web site for dynamic content would primarily be that of a load-reducing alternative to 'redundant' computations by reusing stored results from prior executions, thereby freeing up the strained resources for fresh computations. Factor (1) that determines cache effectiveness now becomes the trade-off between the overheads for storing and managing caches for results produced during dynamic content generation versus the increased throughput and reduced computation time (part of user-perceived latency) from using cached results. Factor (2) now becomes the extent of re-use of the cached computation results during the content generation process for new dynamic content requests.

While increased server loads and latencies are apparent with dynamic content, it is not yet established as to what kind of benefits reusing cached dynamic content computation results can obtain. Questions related to factor (1) that need to be answered are
• What results can/should be cached for effective performance? Where/How should the cached results be stored for effective utilization?

• By nature, results associated with dynamic content change frequently. What mechanisms would be required to keep the cached results consistent? What are the trade-offs for storing and managing the cached results for dynamic content?

• Would the cost of keeping the cached results consistent overwhelm the benefit from their usage?

Similarly, the questions related to factor (2) are

• How 'dynamic' is the actual data delivered as dynamic content? Can it be computed before request-time, so that a stored result can be useful for an incoming request i.e. how cacheable is the dynamic content?

• What results can see significant re-use? Or, what is the extent of re-use for cacheable components?

The first contribution of my work is to establish the feasibility and effectiveness of caching solutions for dynamic content delivery. The caching solutions are all at the web site with a focus to improve the throughput and response times at the web site by decreasing the load on the web site. In this regard, my work provides quantitative answers to the questions concerning the effectiveness of caching solutions as presented above. To do this, I evaluate my caching solutions using a very popular dynamic content application - e-commerce. I use representative e-commerce workloads from the web-benchmark, TPC-W [15], for this purpose. I implement and evaluate the different cache components and the overall strategy
on a web site whose system components for dynamic content delivery consist of
the Apache web server [16], PHP content generation module [17], and the MySQL
database server [18].

The second, equally important contribution is on the nature of caching solutions
that are proposed to address the performance problem. Most of the recently pro-
posed solutions for dynamic content caching (discussed next in section 1.4) attempt
to provide custom caching solutions that combine the caching strategies with the
dynamic content generation tools and mechanisms. In other words, caching benefits
can be obtained but only by using the prescribed methods and tools for dynamic
content generation introduced in their proposals. The apparent technical motivation
behind this approach is to make effective use of the content developer's knowledge of
the application to implement an effective caching solution. Questions such as what
to cache, how to maintain cached components etc are addressed with the developer
specifying them during the coding process for content generation using custom tools
and methods. In my work, the focus is on providing transparent caching solutions
i.e. the solutions should not be tied to any specific tool/method used to create or
program for dynamic content creation. The administrator's or developer's knowl-
dge of the application can still be utilized to make the caches more effective by
using potentially independent set of directives not tied to the content development
tools or methods. This allows the primary performance goals to be attained and at
the same time enables the benefits of caching while using any of the widely deployed
current technologies for dynamic content generation.

The following section presents some of the key related work in the areas of caching
for database applications and dynamic content web sites.
1.4 Related Work

Caching to improve database application performance has been a widely researched area. Related topics includes caching at the clients of data retrieved from servers [19, 20], caching query results in middleware [21], intelligent management and use of the query cache [22, 23, 24] in distributed client-server environments, management of cached and pre-computed results in data warehouse environments [25, 26, 27, 28], and the selection and maintenance of materialized views [29, 30, 31]. Most of these systems consider caching for other applications which have very different characteristics and performance requirements. Also, they just examine a single server-side or client-side cache and not a multi-tiered caching approach as in this work.

Labriniidis and Roussopoulos's work on webview materialization [32] explores two alternatives for the materialization of web views (pages automatically created from base data stored in a DBMS) - either web-page caching at web server or query result caching on database server. They provide theoretical analysis that show the superiority of web-page caching over query result caching and substantiate it empirically with a simple select-based workload for web views. They do not consider a multi-tier caching strategy that might be more beneficial for complex workloads, such as TPC-W.

The Weave management system [33] developed at INRIA can support caching a multiple levels - database data, XML fragments and HTML files. However, it requires a declarative specification of the web site specific to the Weave frame-work for the system to understand the cache-able components of the site. The system does not support transparent caching applicable to a wide variety of contemporary content generation tools unlike the approach proposed in this work.
Challenger et al. [34, 35] propose an approach to track the modifications to cached dynamic content using an object dependency graph that maps the cached objects to the underlying data. In a related work [36], a publishing system for creating/uploading dynamic content to a web-site is presented. By using a template-based design approach different fragments composing the pages are specified. Caching is then performed at the granularity of these fragments. While the characteristics of their algorithms are analyzed, there is no evaluation of the benefits from caching for overall system performance. Their proposal requires the adoption of their template-based content creation framework to obtain the benefits of dynamic content caching.

Candan et al. [37] propose another approach for invalidating cached dynamic content pages when conflicting updates are made to the underlying data. A query-URL map is created from the logs at the web server to keep track of the queries involved in the creation of each cached web page. The update log of the database is then used to identify invalidation events that could make the cached pages stale (by using the invalidated query results and the web pages they are used in from the query-URL map). This method avoids the overheads of triggers on the database system present in Challenger et al.'s approach discussed above. Their system, however, caches only complete pages losing out on the advantages of caching at multiple granularities as proposed in this work.

A related issue is the caching of dynamic content at proxies. Cao et al. [38] propose using server supplied cache applets at proxies to maintain the cached content consistent among other things. Smith et al. [39] use extensions to the HTTP/1.1 protocol for the administrator or web-page designer to specify the validity of the same cached results for different requests. Their approach is not specific to dynamic
content but can be applied to some forms of dynamic content.

Commercial implementations of caches for dynamic content are just beginning to appear. The Oracle 8i cache [40] uses Oracle's database replication facilities to cache entire database tables to answer any database query involving single cached tables. Data consistency is loosely maintained by administrator-controlled cache refresh. This approach is beneficial for workloads with primarily read-only queries involving single tables but could see little benefit and many consistency problems when updates increase. The Dynamai [41] caching server caches complete responses using request- and script-based declarations to map the dependencies between cached pages and external events. The Oracle 9i cache [42] is a complete response cache using HTTP- and time-based invalidation schemes to maintain cache consistency. It can also work with database triggers and programmatic interfaces to refresh cached data when the underlying content changes. In contrast, this proposal advocates a multi-layered caching approach that includes query result caching to obtain the best performance.

1.5 Dissertation Overview

The rest of this dissertation is organized as follows. In chapter 2, I present the multi-tier caching approach for dynamic content caching which improves performance without compromising the freedom to adopt any content generation tool/methodology. The chapter discusses in detail the design and implementation of each of the components in the proposed caching strategy. The next chapter (chapter 3) discusses the workloads from TPC-W that I use to evaluate dynamic content caching and effectiveness of the caching strategy. In chapter 4, I present the experiments and results for the evaluation of dynamic content caching. Chapter 5 has the
conclusions and some of my ideas for future work on related issues.
Chapter 2

Multi-tier Caching of Dynamic Content

The focus of my caching strategy is to lower the load on the servers and help increase throughput and decrease response time at the web site - thus, all the caching solutions address bottlenecks at the web site. Proxy caching for dynamic content addresses issues which are somewhat orthogonal and its benefits are complementary to the solutions from this work. The principle feature distinguishing this approach from other contemporary dynamic content caching work is the adoption of a multi-tier caching strategy that is independent of content development tools and methodologies and can, hence, work with most existing technologies for dynamic content delivery.

A web site serving dynamic content has a multi-tiered architecture - typically three, consisting of the web server for internet communication, a middle-tier or middleware of the application development technologies (which often shares resources with the web server), and a third-tier of additional servers such as a database server providing access to other necessary data or services. With varied and plentiful computation at each tier there is scope for different forms of caching in the web site infrastructure. My caching strategy proposes using an appropriate caching solution to each of the layers in the web site infrastructure for dynamic content.

Caching at the database layer is in the form of query results. This is in addition to the buffer management and table data caching performed in database systems to reduced disk I/O. Query result caching stores the results of prior queries issued to
the database system and re-uses them for future requests of results from the same or related queries. It not only reduces the CPU processing load for the queries but often avoids disk I/Os and memory-cache pollution that result from queries involving large volumes of data - this results in significant improvements in response time in addition to the improvement in database throughput.

At the middle-tier or the application development layer, the dynamic content specification is in the form of a program or script that needs to be executed to produce the content. Before execution, the script needs to be parsed and compiled into a form that can be handled by the execution engine. This becomes critical when it is done at request-time and when the resources are shared between the middle-tier and the web server, as is often the case. Caching the compiled scripts is often a win-win situation since the storage versus re-use trade-offs happen to be quite high for this layer.

At the web server, caching is in the form of the complete web-page that is sent as the response to the dynamic content request. For a request that hits in the cache, this avoids all significant computation at the site - the script parsing and compilation in the middle-tier, the execution of the script and the database querying. The request gets served just like a static file request. On a cache hit, this provides the maximum savings in computation among all the levels of caching.

Caching at all three tiers is implemented using extension modules that can be plugged into the existing system or by minimal changes in the existing dynamic content generation approach. The underlying system infrastructure is left unaltered.

This approach of layer-specific caching solutions through modular extensions to software at each layer accomplishes three important goals:

- The strategy targets all potential computation bottlenecks in the system.
• The strategy accommodates independent implementations of the caching components allowing layer-specific and product-specific optimizations while freeing the overall caching strategy from any dependence on a specific product suite or content creation methodology.

• Cache misses at one tier can still see a cache hit at another tier reducing the miss penalties when compared to a single level complete response cache.

Next, we consider the important issue of maintaining the consistency of cached components - more specifically the policy for maintaining consistency. In the sections following that we look at each caching component in detail - their role in the overall caching strategy, and their design and implementation.

### 2.1 Cache Consistency

The key difference between caching for dynamic content and that for static content arises from the relatively irregular intervals and higher frequency with which cached dynamic content can become stale. Static content typically becomes stale at specific intervals or over a relatively long period in time when new content is published that replaces the old one. The standard mechanism to provide consistency for static documents is by using expiration times. Each document is associated with a Time-To-Live (TTL) value at creation time or upon caching. Once the cached document has been retained past this threshold its copy in the cache is invalidated. Dynamic content on the other hand is typically more fast changing with events causing the changes also being more irregular. Often it is an unpredictable change in the underlying data from which a dynamic web page has been created that causes the page to become stale. Further, the correctness, i.e., having the most current value
of dynamic content object can often be a high priority at many web sites. Thus, when caching dynamic content, additional mechanisms to maintain their consistency become necessary.

In general, cache consistency mechanisms can be classified into two broad categories - invalidation-based and update-based. The overheads to benefit ratio is strongly dependent on the match between the workload and the category of consistency mechanism.

Invalidation-based systems effectively remove cached objects once the event causing them to go stale occurs. Thus, the next access to that object would result in a cache miss. The new object produced by the re-computation can now be cached for future accesses till it in turn becomes stale. With update-based systems, once a cached object becomes stale it is replaced by a new object (that can be used for the same request) in the cache even before any future accesses are made to it. The first access after the object becomes stale would also potentially then see a cache hit. However, in order to create this new object the update-based system may often have to schedule a new re-computation which the system would otherwise not perform.

While update-based systems can see a higher cache hit ratio, it comes at the cost of potentially higher overheads. Re-computation for cached objects are performed at the rate at which they become stale which can be quite different from the rate at which they are accessed. With invalidation, only re-accessed stale cache objects require re-computation. In multi-tiered systems the re-computation often causes an imbalanced increase in load among the different tiers. This imbalance is particularly troublesome if it further increases the existing imbalance (due to the nature of the workload) between the different tiers. Complexity of the implementation is also typically higher for update-based systems which also leads to higher overheads.
All the implementations used in the evaluations utilize invalidation-based cache consistency mechanisms as all the three factors - high rate of invalidating events in some of the workloads, increased imbalance in load between the tiers from the cache updates, and the increased complexity of implementing updates (potentially requiring significant alterations to the system) - favor this choice.

2.2 Query Cache

The query cache (also referred to as the QCache) caches query results in the database system and targets a potential database bottleneck. By re-using results stored in the query cache, re-evaluation of queries can be avoided potentially increasing the scalability of the database system. The benefit from the QCache (and also from any data cache including the page cache) is primarily dependent on two characteristics of the workload: cache re-use and cache hit benefit. The cache re-use of a workload is proportional to the ratio of the read (select) to update queries and the repetitiveness of the read queries. Workloads that have larger values for both would see higher QCache re-use. The repetition of a read query can happen not just from another request for the same response but through a request for any response the generation of which requires the same query to be performed at the backend database. The cache hit benefit for a particular query is proportional to the ratio of the query evaluation time to the cached-result retrieval time for the same query. Workloads that have a higher proportion of queries with large cache hit benefit values are likely to see a greater benefit from the QCache.

Ideally, one would like the query cache to be a transparent add-on module to the database system. However, architectures of database systems are not very amenable to extensions through add-on modules that affect basic operations such as when
and how they evaluate queries. To avoid changes to the database system code and expedite the implementation for the evaluations I use an application-controlled approach to query cache design.

With the application-controlled approach the content generation scripts (written in PHP) are modified to access and maintain the QCache. The QCache consists of a bunch of separate database tables (referred to as cache tables) that are used to store query results. For a query whose result could be in the cache, the content generation script containing it is modified to first check the cache table corresponding to the result. If the cache table contains a valid result for the query, that result is retrieved from the table instead of re-evaluating the original query that produced the result. Only when the cache tables do not have the result is the original query submitted to the database system for evaluation. Figure 2.1 illustrates the code change required to incorporate the QCache with an example. The original select query is given on the left and the ones replacing it for the QCache given on the right. Corresponding to results of this type is the cache table getRelated.ct. So the changed set of queries first look for the result by its key i.id in the query cache. If found that result is returned, else the original query is performed and its result inserted into the cache table. And, finally the result is retrieved from the cache table with another select query. Appendix B contains the complete set of modifications to the queries in the application code for implementing the query cache.

With application-controlled caching, besides the expediency of the implementation one also obtain the benefit of getting the performance of the best query cache implementation. One can use site-specific and workload-specific knowledge to utilize the query cache only for beneficial queries. Queries that do not have any significant cache re-use or cache hit benefit can be avoided making efficient use of the QCache.
Original query:

```sql
SELECT J.i.id, J.i.thumbnail
FROM item I, item J
WHERE (I.i.related1 = J.i.id or I.i.related2 = J.i.id) and I.i.id = <input>;
```

For Query Cache:

```sql
SELECT i.related, i.related.thumbnail
FROM getRelated_ct WHERE i.id = <input>;
```

If result not in cache table:

```sql
INSERT INTO getRelated_ct SELECT J.i.id, J.i.thumbnail, <input>
FROM item I, item J
WHERE (I.i.related1 = J.i.id or I.i.related2 = J.i.id)
and I.i.id = <input>;
```

```sql
SELECT i.related, i.related.thumbnail
FROM getRelated_ct
WHERE i.id = <input>;
```

Figure 2.1: Example of the application code change to incorporate QCache
resources. The scheme works similar to having an SQL extension such as `select cached` available to the programmer in addition to the normal `select` querying semantics. The `select cached` query would indicate to the system the programmer preference that the result of the query specified be cached and be served from the cache. For caching queries with bounded number and size of result sets I use in-memory database tables. For queries with an unbounded number of result sets I use disk-resident database tables.

As discussed earlier (section 2.1), I use an invalidation-based consistency mechanism. In the evaluation workloads, all updates to the system also take place through dynamic content requests to the web-site. Hence, in the scripts that contain these updates I also insert appropriate cache invalidation calls. Appendix B contains the additional commands to the database system to perform the cache invalidations. Synchronization for the accesses to the cache tables is provided using calls to the table locking mechanisms of the database system, thus, using the existing database infrastructure to implement the QCache.

In chapter 5, I present some of the issues in the implementation of a transparent query caching solution that can be either a stand-alone proxy or integrated with the database system. Such a solution would not require application code modification for its implementation.

### 2.3 Script Cache

The script cache targets the web server bottleneck. By caching the compiled scripts that are invoked to answer dynamic content requests, the interpretation overheads in dynamic content generation are avoided. The script cache re-use is typically equal to the repetition in the script requests to the server. In a typical dynamic-content
oriented site a few key scripts generate the different user-specific content based on
differences in the values of request-specific parameters. Hence, the number of scripts
to cache is invariably small enough even to fit in memory (resulting in no misses in
the compiled script cache for want of space to cache them). In addition, cached pages
become stale only when the original script is changed or the site re-organized. For
most sites, in particular e-commerce sites, this is much less frequent than updates
to backend databases that could invalidate objects cached in the other tiers. Thus,
the cache re-use for the cached pages would be close to 100%. As content and
services offered by sites become more complex, the interpretation overhead typically
increases even with increasingly efficient script engines. Thus, the script cache is
almost always a useful cache tier.

For the PHP script engine that I use, there are many publicly available script
cache packages. I use the open-source, script cache package for PHP [17], Alternate
PHP Cache (APC) [43]. The cache is built as an extension module to the PHP
script engine. The package supports both a shared memory implementation as well
as a memory-mapped one. With no significant difference in their performance on
the FreeBSD platform I use, I went with the memory-mapped version. Calls to the
regular script engine's interpreter function which is called when any script page has
been invoked is replaced by an APC function call. This function call checks for
the presence of a cached compiled version of the script first. If present (and is still
valid) the cached version is executed. Otherwise, the regular interpreter is invoked
and a compiled version created, which is then used for both the current execution
as well as cached for the future. While APC provides API extensions to maintain
the consistency of the script cache, since there are no updates to the script during
the experiments (which can potentially invalidate objects in the script cache) I do
not make use of any of them.

The overheads for script caching are minimal. The storage overhead is dependent on the number and size of dynamic content generation scripts. As pointed out earlier, typically a few script pages suffice to answer a number of different requests. Thus the number of different objects that need to be cached are usually much fewer than for the other tiers. Further, this small storage overhead is the main overhead for script caching - with low update rates, there is very little contention overheads in any reasonable implementation.

2.4 DCache - The Complete Response Page Cache

The complete page cache component (also referred to as DCache later) alleviates both the web server and database server bottlenecks. By caching complete page responses, a cache hit is converted to a 'static page request' from that for a dynamic page. All the computation and overheads associated with a dynamic page request - script compilation and execution, communication with other systems (e.g. database system), and any processing in the backend systems such as query evaluation in the database system - are completely avoided. Thus, it is reasonable to expect maximum benefits for a cache hit at this tier. The overall performance from caching, however, depends on additional factors. The cache re-use is typically lower than at the other two tiers. Every page cached needs a completely identical request to see a cache hit, unlike the cross-page re-use of query results with QCache (section 2.2) or the re-use of the same cached code to generate multiple different responses in the case of the script cache. Further, the cache-ability of a response depends on the least cache-able component in it. For example, responses that have components involving updates at the backend database cannot be cached at this tier.
My approach to dynamic content caching integrates the caching technology with the web server. Whenever, any request for dynamic content arrives at the web server it gets transparently handled by the caching module which is part of the web server. To do this, I provide the caching subsystem as an add-on-module for the web server. This caching approach could work with any general-purpose web server. However, to better understand the details of our implementation it is necessary to have a basic knowledge of the architecture of the Apache web server for which I built the module.

2.4.1 Architecture of Apache

Here, I provide a brief introduction to the architecture of Apache [16, 44]. On Unix systems, Apache uses a multi-process model to multiplex its operations. A single parent process is responsible for supervision, while all the child processes participate in serving incoming requests. The request handling task is broken down into multiple phases (nine in the current version). The core server code can handle basic functionality for each phase of a HTTP request. However, extension modules are used to further customize how Apache functions for each of these phases. The modules do so by registering short code routines called handlers that Apache would invoke at the appropriate moment. A module can register handlers for multiple phases. And multiple modules can register handlers for the same phase. Apache invokes the multiple handlers for a phase in the reverse order in which they are registered i.e. the handler that is registered last would be invoked first. If no modules are registered for a particular phase, it will be handled by a default routine from the Apache core.

The caching module, DCache, registers a handler for the content delivery phase
last, for all request types that require dynamic content e.g. pages with server-side scripting like Perl, PHP etc. This means, that whenever a request enters the content delivery phase for a page requiring dynamic content, Apache would first invoke the handler registered by the DCache module.

Apache also provides the facilities for modules to invoke internal requests and subrequests. An internal request is processed almost like any other request, but is actually a request generated within Apache. A subrequest is a special form of internal request. At any phase of request handling, a handler can create a subrequest for a file or URI. Other handlers would service this subrequest just as if it came from a client outside the server. After the subrequest is serviced the control is then returned to the invoking handler.

The subrequest facility allows the DCache module to service any request for dynamic content by using pre-registered handlers of other modules for that particular content type. Once the DCache module intercepts (or receives first) the request for a dynamic content page, it can invoke the handler registered by a module for that content type with a subrequest to obtain the fresh content/response for that request.

2.4.2 DCache – The Cache Module for Dynamic Content

DCache provides dynamic content caching by caching responses to previous requests for the same content. Requests are classified into two broad categories:

- Cacheable requests – For this category, if a previous response has been cached by DCache, it is served from the cache. If not, it makes a subrequest that will be served by the appropriate handler. Once the subrequest returns, the DCache module gathers the response (generated during the subrequest) and
caches it. A request cached by the DCache module can have an expiration time. If so, once the expiration time has elapsed, any new request for that content results in a subrequest to obtain a fresh response for that request.

- Non-cacheable request – For this category, the content handler of the DCache module just *declines* the request. This results in Apache invoking the previously registered (next in priority) handler of another module or an apache core routine (in the absence of any module registering a handler) for the content handling phase for that request.

Once the response for a request gets cached it is likely that the cached response would turn out to be invalid for the request at a later point in time – basic nature of dynamic content. DCache supports two procedures for invalidating cached responses. One is through expiration times and the other is through a process we call *request-initiated* invalidation. The first is straight-forward: when the response is generated for a dynamic content request, the appropriate handler also generates an expiration time for the request. This information is passed to the DCache’s content handler through the headers generated for the subrequest. DCache then stores this expiration time along with the cached response. When a *cacheable* request arrives after the expiration time has elapsed, DCache generates a new subrequest for it, instead of returning the cached response.

*Request-initiated* invalidations support the situations where the request for a specific page A causes an update in the database system which renders stale the data cached for another page B. For request-initiated invalidation the site builder registers rules with the DCache handler to identify cached responses that are made stale by the servicing of a specific request – these can be in the form of simple if-then rules e.g.
if request is for document A – with args x,y,z

then invalidate cached responses B, C, ..

Whenever a new request is received, whether it belongs to the cacheable or non-cacheable category, in addition to handling the specific request, any request-initiated invalidations registered for it are also carried out by the DCache module. The subsequent request for an invalidated response would result in a fresh content being generated for it.

2.4.3 DCache implementation

This section details the current implementation of the actual cache. The multi-process nature of the Apache server and the support of the specific Unix platform (FreeBSD) strongly influences this design of the cache. Given the multi-process nature of Apache, we need a cache that should be shareable between multiple processes. This is to make the response cached by one Apache process available for use by another Apache process. The two obvious choices to obtain this sharing between processes is – shared memory and memory mapped files. There is little to choose between the two in terms of access overhead in FreeBSD.

Memory-mapped Cache

I go with memory mapped files to share the cached responses as it makes it easier to implement the DCache as a transparent module that intercepts the dynamic content requests. Apache supplies a file descriptor to each module's content handler to which the handler writes the response. This file descriptor is normally that of the connection to the client and, thus, the response written by the handler is directly
sent to the client. The DCache replaces this file descriptor with that of the memory mapped file when making the subrequest (as explained in section 2.4.1). Thus, the 'real' handler for a particular dynamic content type invoked by the subrequest just writes the content it generates to the file descriptor of the memory mapped file instead of the one for the client connection. It does not need to know the presence of the DCache module to supply it with the response to cache. Once the response has been written into the memory mapped file, it is sent to the actual connection by the DCache module. The transparency of the DCache's operations to the rest of the system makes it a transparent add-on to the rest of the system enabling it to be used with any content generation module.

The cache is composed of an in-memory hash table and memory mapped files that contain the cached responses. There is a fixed number of buckets in the hash table, one for each cached response. The cached response corresponding to each hash bucket is backed by a specific memory mapped file. A hash function maps the URL (with the arguments if necessary) of the cache request to a specific hash bucket. This hash function can be site-specific for better characteristics (a default hash function is also supplied with the system).

The hash table also needs to be shared between the Apache processes. This is done, again, by having it backed by a memory mapped file. However, the mapping is done such that the updates to the table are asynchronously reflected to disk, thus retaining the performance advantage of an in-memory hash table.

**Synchronization**

We need efficient synchronization between the multiple apache processes when they attempt to concurrently read (write) to the cached responses or to the hash table.
I use efficient atomic reader-writer locks implemented with code borrowed from the FreeBSD source. There is a lock per hash bucket to eliminate unnecessary synchronization overheads. The locks themselves are shared by allocating them at the beginning of the shared mmap'd file for the hashtable.

**Cache Replacement**

As a result of the cache design, we have the flexibility to have a different replacement policy per response. Since the hash-function can be site-specific (user-supplied) it can regulate what set of responses get mapped to what hash buckets. High-priority (frequently requested) pages can have their own hash buckets, while multiple lower priority pages can be hashed to the same hash bucket. This is a simple but very effective mechanism to provide a differentiated cache priority scheme to account for the different levels of request priority that a web-site might like to support.
Chapter 3

TPC-W - Evaluation Workloads

This chapter presents the workloads used in evaluating the performance of the proposed caching strategy and techniques. The workloads are from the TPC-W benchmark provided by the Transactional Processing Council [15]. TPC-W has been specifically designed to evaluate complete systems supporting web-commerce. It includes detailed specifications for a controlled environment simulating the activities of an online retail book store. The specifications detail the interactions between the client and the web site, the activities on the web site for each interaction, the corresponding response, and the database content at the site with the changes induced in them during each interaction.

The benchmark consists of three different workloads - browsing, shopping, and ordering - each consisting of a different proportion of the same 14 different interactions. Table 3.1 lists the interactions and their proportions in each workload. The interactions are grouped in two categories - browse and buy - containing interactions pertaining to online browsing and buying, respectively. The three workloads have different browse to buy ratio to emulate the activities of three common kinds of web-commerce sites. The interactions in the browse category involve just read queries to the web site, while the ones in the buy category can cause many updates to the backend database.

In the following section, I present the characteristics of the interactions and each of the workloads in TPC-W from the view point of caching. In the next section I
<table>
<thead>
<tr>
<th>Web Interaction</th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browse</td>
<td>95%</td>
<td>80%</td>
<td>50%</td>
</tr>
<tr>
<td>Home</td>
<td>29.00%</td>
<td>16.00%</td>
<td>9.12%</td>
</tr>
<tr>
<td>New Products</td>
<td>11.00%</td>
<td>5.00%</td>
<td>0.46%</td>
</tr>
<tr>
<td>Best Sellers</td>
<td>11.00%</td>
<td>5.00%</td>
<td>0.46%</td>
</tr>
<tr>
<td>Product Detail</td>
<td>21.00%</td>
<td>17.00%</td>
<td>12.35%</td>
</tr>
<tr>
<td>Search Request</td>
<td>12.00%</td>
<td>20.00%</td>
<td>14.53%</td>
</tr>
<tr>
<td>Search Results</td>
<td>11.00%</td>
<td>17.00%</td>
<td>13.08%</td>
</tr>
<tr>
<td>Order</td>
<td>5%</td>
<td>20%</td>
<td>50%</td>
</tr>
<tr>
<td>Shopping Cart</td>
<td>2.00%</td>
<td>11.60%</td>
<td>13.53%</td>
</tr>
<tr>
<td>Customer Registration</td>
<td>0.82%</td>
<td>3.00%</td>
<td>12.86%</td>
</tr>
<tr>
<td>Buy Request</td>
<td>0.75%</td>
<td>2.60%</td>
<td>12.73%</td>
</tr>
<tr>
<td>Buy Confirm</td>
<td>0.69%</td>
<td>1.20%</td>
<td>10.18%</td>
</tr>
<tr>
<td>Order Inquiry</td>
<td>0.30%</td>
<td>0.75%</td>
<td>0.25%</td>
</tr>
<tr>
<td>Order Display</td>
<td>0.25%</td>
<td>0.66%</td>
<td>0.22%</td>
</tr>
<tr>
<td>Admin Request</td>
<td>0.10%</td>
<td>0.10%</td>
<td>0.12%</td>
</tr>
<tr>
<td>Admin Confirm</td>
<td>0.09%</td>
<td>0.09%</td>
<td>0.11%</td>
</tr>
</tbody>
</table>

Table 3.1: Interactions Mix for TPC-W Workloads

present some pertinent details of my implementation of TPC-W.

3.1 Cache Characteristics for TPC-W Workloads

The interactions in the two categories, browse and buy have very distinct characteristics. The browse group has only read queries that make no changes to the data in the system. The buy group of interactions have a number of writes besides reads. All the write queries update or insert individual rows or fields in the database tables - and hence are typically of short duration. However, their dependence on the read values and the exclusive access they need for the writes could slow down the write queries. Among the read queries, on the other hand, are some fairly expensive queries - particularly in the BestSellers interaction and the SearchResults (depending on the database size). As a class, the browse group of interactions put a greater
load on the servers than the buy group.

With updates dominating the interactions in the buy category, there is little that is cache-able in those interactions. For the page cache, this implies that the responses for most of those interactions (all except Customer Registration and Admin Request) are not cache-able. For the query cache, only a few of the queries in the buy category are cache-able. The implication of this is that the ordering workload with a significant proportion of interactions from the buy category should see reduced benefits from both page cache and query cache. The benefits should be maximum for browsing and slightly lower for shopping.

Five of the interactions - Home, New Products, Best Sellers, Search Request, and Search Result - contain an ad component that is required to be randomly chosen at request time. The inclusion of the logic for the random choice of ad in the page creation process renders the responses uncachable. The conventional approach to incorporate random ads typically involve requests for embedded content from separate ad servers leaving the logic for the main content creation unaffected and the response cacheable. To enable similar caching of responses possible with conventional ad-inclusion techniques, I relax the condition of choosing a random ad for every response by applying a small non-zero expiration time (default is 2 minutes) to responses that contain the ad component.

In addition to expiry of cached pages, the Buy Confirm and Admin Confirm interactions contain update queries which change the backend database in such a way that some of the cached objects in both the DCache and query cache can go stale. All invalidations that occur from such interaction behavior are effective immediately and are implemented using the strategies mentioned in the previous section for each cache tier. The script cache can cache the compiled scripts for all
<table>
<thead>
<tr>
<th>Table Name</th>
<th>Cardinality</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>2,880,000</td>
<td>1,316,417</td>
</tr>
<tr>
<td>Address</td>
<td>5,760,000</td>
<td>781,657</td>
</tr>
<tr>
<td>Orders</td>
<td>2,592,000</td>
<td>297,297</td>
</tr>
<tr>
<td>Order Line</td>
<td>7,782,313</td>
<td>875,465</td>
</tr>
<tr>
<td>CC_Xacts</td>
<td>2,592,000</td>
<td>253,185</td>
</tr>
<tr>
<td>Item</td>
<td>10,000</td>
<td>6,834</td>
</tr>
<tr>
<td>Author</td>
<td>2,500</td>
<td>1,145</td>
</tr>
<tr>
<td>Country</td>
<td>92</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 3.2: TPC-W Database Tables

the interactions - further as the scripts do not change during the experiments, the script cache entries never get invalidated.

Appendix A lists all the queries in each interaction of TPC-W.

3.2 Implementation

Table 3.2 lists the database tables specified in TPC-W and their sizes for the chosen scale of implementation at the beginning of each run. The TPC-W benchmark lists two parameters that specify the scale of database - the number of items in the Item table (which determines the number of entries in the Author table too) and another scale factor (number of emulated browsers) which affects the number of Customer table entries. The sizes of all other tables in the database except the Country table are related to the size of the customer table. For the base database configuration show in table 3.2, I use 10,000 for the number of items and 1,000 as the scale factor determining the customer table size (resulting in its cardinality of 2,880,000).

The Size column in table 3.2 includes the storage space for both the data and the indices required to optimize query evaluation times. There are two other tables used during the execution of the workloads that deal with the emulation of the
customers' shopping carts - each execution of the workloads is begun with these two tables empty. In addition to the data in the database tables, there are about 180 MB of images on the web server representing the multimedia content for the web site.

Each of the interactions is implemented as a PHP [17] script, resulting in 14 different php files for the TPC-W benchmark. PHP provides client API calls to the backend database server through which appropriate database queries can be sent and their results retrieved. I also use its persistent database connection API to reduce connection overheads between web server and database server.

The client requests to the server are specified in sufficient detail in TPC-W with through by giving the interaction-transition probabilities for all pairs of interactions. Each emulated client represents a user session to the server during which the client would request for a sequence of interactions and along with the embedded images (representing the multimedia content) in each interaction response. TPC-W recommends a mean emulated user-session length of 15 minutes and a mean interaction think time of 7 seconds - these are the values used in the experiments. The mean think time specification places an upper bound on the maximum throughput that can be obtained for a given number of emulated concurrent clients. For example, while emulating 100 concurrent clients the maximum throughput that can be obtained is 100/7 = 14.29 interactions/second.

I differ from the TPC-W specifications in the following. I do not implement secure communications and authentication layers required for purchase transactions. Since they form only a small percentage of transactions (which even otherwise see minimal benefit from most caching approaches because they include updates), I believe that the effect of this on the results is minimal. MySQL which I use as the database software does not fully support the required ACID semantics for transac-
tions. I support the required atomicity and isolation semantics using the database's locking mechanisms. With a system supporting complete transaction semantics the database overheads are likely to be higher, thus, placing an even greater importance on caching benefits.
Chapter 4

Evaluation of Caching in Database-driven web sites

This chapter presents the experiments and results for the evaluation of the caching techniques. First is the experimental set-up. Following that is the presentation of the base system performance without any caching. After that are the caching results - they are divided into three sections. Section 4.3 presents the main results for caching - the benefits from each caching component and the synergistic nature of the components evaluated with the results from combining all caching components. Section 4.4 presents the effect of caching on an alternate platform - this shows the effect of system configuration on performance. Section 4.5 shows the results with different configuration for the TPC-W workloads. Finally, in section 4.6 is a summary of all the evaluation results.

Performance comparisons are made based on the values for two parameters: throughput expressed in interactions per minute and the average LAN response time for each interaction. The LAN response times are preferred over response times that include emulated network delays because this gives a more accurate picture of caching's impact on the servers which is this work's main focus. Further, for dynamic content workloads server-response time often dominates network delays particularly under high server loads. Hence, the LAN response time is a reasonable indication of client-perceived delay too. Where pertinent, the response time is also broken down into three components - the time spent in the database server, time
spent in the scripting engine and the time spent in the web server core (estimated as the remaining time in the measured response time).

4.1 Experimental Setup

All the experiments are performed with three machines - one for the client emulator, a second for the web server and the third for the database server. Both the servers have 1.33Ghz AMD Athlon processors and 30G 7200 r.p.m disks. The web server has 1.5GB of memory while the database server has 768MB of memory. We found no virtual memory paging behavior on the database system to warrant adding extra memory for any of the runs. The client emulator has an 800Mhz Athlon processor with 256MB of memory - this was found adequate for emulating the highest loads in the experiments without proving to be a bottleneck. The operating system on all the machines is FreeBSD. The machines are connected by a 100Mbps Ethernet LAN.

Apache v.1.3 [16] is the web server. Being the most widely used web server with good support for add-on modules, it was an obvious choice. It is configured with the PHP v.4.0.3 [17] module for server-side scripting to generate dynamic content (including communication with the backend database). PHP is Apache’s most popular add-on module (source [45]) for dynamic content and database connectivity. MySQL v.3.23 [18] is the database server. MySQL is used instead of more established database systems from DB2, Oracle, Microsoft, or Informix because of its open-source nature. This enables us to examine it in much greater detail and understand the system behavior. Additionally, it is one of the most popular backends for database-driven web sites, partly due to its impressive performance for such workloads. One of the key difference between MySQL and other established
systems is its usage of regular files for database tables and reliance on the operating system file cache for data caching. In all the experiments there is little or no page out activity associated with bad data caching - may be a common characteristic of e-commerce workloads - thus, it has no bearing on the results.

For each run, the client program emulates a fixed number of concurrent clients to the web server. For each emulated client, it goes through cycles of starting a session, sending interaction requests to the web server, collecting responses and examining them, and terminating the session (and connection). The client program also collects much of the statistics either by its own measurement or from measurements made on the servers in the caching modules and passed on as part of the responses from the web server. Each evaluation run lasts fifteen minutes with statistics averaged over that period. There is an additional five minutes allowed at the beginning of each run to discard cold start effects. The caching runs have pre-populated caches from a 2 hour run to emulate steady state behavior while allowing the results to be obtained in a reasonable amount of time.

4.2 Base System Performance

This section presents the the base system performance, i.e. the system without any caching, for the three workloads of TPC-W. Figure 4.1 shows the variation of the throughput (in interactions per minute) and response times (in seconds) with the number of concurrent clients emulated in the experiments. For all three workloads, as the load increases the throughput increases significantly and the response time very gradually, initially. However, after the peak throughput has been reached there is a much sharper increase in response time accompanied by a gradual decline in throughput. This occurs because a bottleneck has been reached in the run and
Figure 4.1: Performance of the Base System Without any Caching
Figure 4.2: Response Time Components at Peak Throughput

Further increase in system load cannot result in more work being done on the server. The number given besides the throughput and response time curves give their values at peak throughput. Note that the scale on the x-axis that shows the number of clients varies for the three workloads. This is so as the range where the figures exhibit interesting behavior differs between the workloads.

The figure also highlights the difference between the three workloads. Browsing has the lowest throughputs and the maximum response times for a given number of concurrent clients indicating that it puts the maximum load on the servers. It also reaches its peak throughput at the lowest number of concurrent clients - 80. Shopping has better performance with higher throughputs and lower response times, reaching its peak throughput at 140 concurrent clients. Ordering has the best performance. The numbers are in line with nature of the workloads (described in chapter 3) - browsing has the highest proportion of heavy queries and ordering the lowest with shopping in between.

Figure 4.2 shows the normalized breakdown of the average interaction response
<table>
<thead>
<tr>
<th>Workload</th>
<th>Load (number of clients)</th>
<th>Throughput (interactions/minute)</th>
<th>Response Times (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>DB</td>
</tr>
<tr>
<td>Browsing</td>
<td>80</td>
<td>9716</td>
<td>428</td>
</tr>
<tr>
<td>Shopping</td>
<td>140</td>
<td>16896</td>
<td>647</td>
</tr>
<tr>
<td>Ordering</td>
<td>380</td>
<td>39596</td>
<td>1462</td>
</tr>
</tbody>
</table>

Table 4.1: Load, Throughput, and Response Times at Peak Performance of Base System

time into three components - time spent in the database system, time spent in the script engine, and the time spent in the web server core. The component-wise breakdown is given for the response times, again, at peak performance. For a workload without dynamic content the web-server component alone would be there. Thus, the size of the web-server component indicates how close the particular workload is to a pure static content workload. This breakdown, again, illustrates the difference between the three workloads. It shows that for both browsing and shopping workloads the database component dominates, accounting for over 90% of the response time. Ordering, has significant script engine and noticeable web-server components with the database component still accounting for a dominant 66% of the response time. Table 4.1 gives the values for the load, throughput and response times for each of the three interaction mixes at their peak throughput.

In the following section, the peak base system performance for each workload would be compared with the caching performance at corresponding loads and with the peak caching performance.
4.3 Caching Performance

In this section, we look at the performance of the individual caching components and the overall caching strategy on the TPC-W workloads - browsing, shopping and ordering. The caching system's performance is compared with the peak performance of the base system shown in the previous section.

4.3.1 Query Cache

The query cache or QCachge targets the database overheads for read queries. Reducing the number of read queries performed on the server can also speed up the write queries due to reduced server loads and lower locking overheads for accessing the data. Thus, the QCachge is expected to reduce the load on the database server increasing overall performance. The QCachge, as implemented, can impose an additional burden on the web server. The modified application scripts are slightly longer incurring additional script parsing and compilation costs. Where the workload has little query result re-use it will incur additional overheads for redundant checks on the cache tables.

The performance of the query cache for the three workloads - browsing, shopping and ordering are given in figures 4.3, 4.4, and 4.5, respectively. All the figures present the QCachge performance at the load (number of concurrent clients) at which the base system gets its peak performance and at the load at which QCachge gets its peak, comparing both with the peak performance of the base system. In the figures, NC-X, denotes the peak performance of the base system (no cache) - this peak performance is obtained at X number of clients; and QC-Y represents the QCachge performance at Y number of concurrent clients (one bar where Y = X, and the other where Y = load for QCachge's peak).
NC-80 - No-Cache Peak, attained with 80 clients.
QC-80 - QCache statistics at 80 clients.
QC-280 - QCACHE Peak, attained with 280 clients.

Figure 4.3 : Performance of Query Cache for Browsing

The QCACHE produces impressive improvement in performance. First comparing the figures for NC-80 and QC-80, we find that response time reduces to 10% of the base system's with query cache. There is little change in throughput as rate of requests is now the limiting factor for throughput. Once that is increased, we see that QCACHE attains its peak performance at 280 concurrent clients (QC-280). The increase in throughput is over three-fold. The increased load on the servers (280 clients versus 80), however, results in a higher latency of 866ms as opposed to 428ms which is the the base system's latency at its peak performance.

The breakdown of the response time into its components indicate that the web server has become the new bottleneck with under 10% of the time being spent in the database system with QC-280. Thus, the QCACHE completely eliminates the database bottleneck, and the higher connection rate drives up the load on the web server (both due to script execution and basic web server activity) making it the new bottleneck.
Figure 4.4 presents a similar picture for the shopping workload. The gains, however, are smaller than for the browsing workload because of the reduced cache re-use with shopping (as indicated in chapter 3). The reduced cache re-use stems from both lesser number of requests with cacheable queries and a greater number of requests that can invalidate cached results. At 140 clients, QCach reduces the response time to under 13% of the base system’s and at 280 clients it obtains a peak performance which is about 1.9 times the peak performance of the base system. Running at twice the number of clients this results in a loaded web server increasing the response time by about 23% over the base system’s.

Figure 4.5 presents the QCach performance for ordering. Here, the QCach actually has lower throughput than the base system at its peak. The peak for the QCach system is again with 280 clients, and at the base system’s peak load of 380 clients the already loaded web-server is under overload conditions resulting in a 127% increase in response time. At the QCach’s peak throughput, which is about
<table>
<thead>
<tr>
<th></th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NC</td>
<td>$QC_{NC,rt}$</td>
<td>QC</td>
</tr>
<tr>
<td>Concurrent Clients</td>
<td>80</td>
<td>240</td>
<td>280</td>
</tr>
<tr>
<td>Throughput (int./min.)</td>
<td>648</td>
<td>1985</td>
<td>2074</td>
</tr>
<tr>
<td>Response Time (ms)</td>
<td>428</td>
<td>272</td>
<td>866</td>
</tr>
</tbody>
</table>

NC - No-Cache Peak
$QC_{rt,NC,rt}$ - QCache statistics with response time comparable to No-Cache Peak’s
QC-280 - QCache Peak

Table 4.2: Query Cache Throughput Comparison At Same or Lower Response Times

21% lower than the base system’s, there is 46% reduction in the response time. The results are in tune with the analysis of the workloads in chapter 3. Ordering has much lower re-use of query results benefiting only a little from cached results while its high rate of updates increase the overheads for maintaining consistency of cached results resulting in a net decrease in overall throughput.

Table 4.2 presents the numbers showing increase in throughput with the same or lesser response time as those at the peak throughput of the base system - these are indicated as $QC_{NC,rt}$. The NC and QC columns show the corresponding values at the peak performance for the base system without caching and the system with query caching, respectively. For e.g. the average response time at the peak throughput for the base system is 428ms for the browsing mix. For the query cache with 240 clients the average response time is 272ms which is significantly less than the 428ms for the base system while the throughput is 1985 interactions/minute which is more than 3 times the base system’s peak throughput of 648 interactions/minute. The table also gives the peak throughput of the system with query caching, for reference.

Table 4.3 summarizes the QCache’s hit and invalidation rates for browsing,
NC-380 - No-Cache Peak, attained with 380 clients.
QC-380 - QCache statistics at 380 clients.
QC-280 - QCache Peak, attained with 280 clients.

Figure 4.5: Performance of Query Cache for Ordering

shopping, and ordering. The important numbers that bring out the differences between the workloads are the *Lookups per Interaction* and *Attempted Invalidation* per minute. Both the decreasing order of values for *Lookups per Interaction* and the increasing order of values for *Attempted Invalidation* per minute clearly support decreasing cache performance in the order of browsing, shopping, and ordering. The hit rates (hits/lookup) are comparable as the nature of results being cached are the same for all three workloads. The *actual invalidation rate* is dependent on whether the results are cached or not when the invalidations are attempted - hence, the discrepancy between attempted-invalidations rate and the actual-invalidations rate. The work involved for the invalidations is dependent on both.

To summarize, the QCache completely eliminates the database bottleneck. This results in improved performance for browsing and shopping. For ordering which has much less cache re-use, it still reduces the database component thus reducing the response time at the peak. However, the increased load on the web server results in
<table>
<thead>
<tr>
<th>Cache Statistic</th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit Rate</td>
<td>0.975</td>
<td>0.961</td>
<td>0.956</td>
</tr>
<tr>
<td>Lookups per Interaction</td>
<td>1.279</td>
<td>1.070</td>
<td>0.649</td>
</tr>
<tr>
<td>Attempted Invalidations per minute</td>
<td>22.600</td>
<td>32.533</td>
<td>221.933</td>
</tr>
<tr>
<td>Actual Invalidations per minute</td>
<td>21.467</td>
<td>26.200</td>
<td>14.933</td>
</tr>
</tbody>
</table>

Table 4.3: Query Cache Statistics at Peak Performance for TPC-W Workloads

lower throughput than the base system.

4.3.2 Script Cache

This section presents the results for the Script Cache for all the three workloads. The script cache should lower the load on the web-server by eliminating the script parsing overheads. For all three workloads the database component dominates in the base system. Since the script cache does not address this bottleneck the system with script cache also attains its peak performance at the same number of concurrent clients to the site. Hence, the figures for script cache performance show the comparison between just two configurations, the base system at its peak and the peak script cache performance also attained with the same number of concurrent clients.

Figures 4.6 and 4.7 indicate the same - script cache has no improvement for these two workloads which are limited by database system performance. In fact, there is a small increase in the database component (seen in the response time breakdown graph) because of the elimination of the script parsing overheads. For the ordering workload, however, there is a noticeable improvement in performance particularly
NC-80 - No-Cache Peak, attained with 80 clients.
SC-80 - Script Cache Peak, attained with 80 clients.

Figure 4.6 : Performance of Script Cache for Browsing

in the response time - it reduces by 39%. Ordering allows a much higher throughput than the other workloads resulting in a higher load on the web server. By addressing the web server bottleneck the script cache improves the performance for ordering. Overall, the results for the base system indicate marginal or no improvement with the script cache.

Figure 4.9 gives the performance of query cache when the script cache is also added. The results there indicate the true value of the script cache. It was shown in the section 4.3.1 how the presence of the query cache could result in the web server becoming the new bottleneck at peak performance. Figure 4.9 shows how the addition of the script cache at this juncture alleviates this bottleneck. The reduction in the Script component of the response time also indicates the source of improvement. There is only a marginal increase in throughput but a dramatic reduction in response times that ranges from 83% (ordering) to 91% (browsing) - at this point neither server is fully utilized and the throughput is limited by the
NC-140 - No-Cache Peak, attained with 140 clients.
SC-140 - Script Cache Peak, attained with 140 clients.

Figure 4.7: Performance of Script Cache for Shopping

NC-380 - No-Cache Peak, attained with 380 clients.
SC-380 - Script Cache Peak, attained with 380 clients.

Figure 4.8: Performance of Script Cache for Ordering
B - browsing, S - shopping, O - ordering  
Q-280 - Query Cache Peak, attained with 280 clients for all workloads.  
QS-400 - Query Cache with Script Cache Peak, Statistics for 400 clients (shopping and ordering).

Figure 4.9 : Performance of Query Cache with Script Cache

rate of incoming requests. The peak performance for the combination of the script cache with query cache indicates just over 50% improvement in throughput for all the workloads with response times that are comparable or lower than the response times at the query cache’s peaks. While the web server is still the bottleneck it is so at significantly improved performance. These results indicate that the contributions of the script cache become significant once the web server becomes the bottleneck in the system.

4.3.3 Page Cache

The Page Cache or DCache eliminates all dynamic content overheads producing the maximum benefits on a cache hit. Figures 4.10, 4.11 and 4.12 show its respective performances for browsing, shopping, and ordering.
For browsing, with 80 concurrent clients that yields the peak throughput for the base system, the system with DCache is very lightly loaded producing an 87% reduction in the response time. At its peak performance, the DCache system’s throughput is 5.8 times that of the base system. This vastly higher throughput is attained at 480 concurrent clients which adds to the web server overheads resulting in 57% increase in the response time at the peak performances. At a lower load (with 440 clients - not shown in the figure), the DCache obtains a throughput of 3575 interactions/minute (5.52 times the base system’s peak) with a response time of 373ms which is lower than the response time at the base system’s peak throughput (428ms).

The breakdown of the response time components indicate that both the database and scripting overheads are being reduced significantly by the DCache. The significant increase in the web server portion indicates the workload is becoming more akin to a static content workload. The web server component also captures time spent in the DCache module - when multiple requests arrive for the same read for a page that is not cached, while the first one is being processes the rest would wait in the DCache adding to the web server component. With the DCache at its peak, the performance is again dependent on the database component. All requests that miss in the cache and the increased number of updates (from the increased throughput), both load the database server again limiting the peak performance at the shown level.

For shopping, there is a 88% reduction in response time at the base system's peak performance load point (140 clients), while the peak throughput attained is 1.95 times that of the base system with twice the number of concurrent clients. The much smaller reduction in the database component as compared to browsing results
NC-80 - No-Cache Peak, attained with 80 clients.
PC-80 - Page Cache Statistic, attained with 80 clients.
PC-480 - Page Cache Peak, attained with 480 clients.

Figure 4.10: Performance of Page Cache for Browsing

from lower frequency of requests for cacheable responses. The graph indicates that the workload is still bottlenecked at the database system.

For ordering, the peak performance for DCache is at 400 clients, very close to the load for peak performance of the base system - 380 clients. Thus, DCache's performance at both load points is very close and marginally higher than the base system's. The improvement shows up more in the response time. There is much less improvement than the other two workloads as there is much less cache usage for ordering.

Table 4.4 summarizes the page cache's hit and invalidation rates for browsing, shopping, and ordering. As with the statistics for the query cache, the decreasing order of values for lookups per Interaction and the increasing order of values for attempted invalidations per minute clearly supports decreasing cache performance in the order of browsing, shopping and ordering. The hit rates (hits/lookup) are comparable as the nature of results being cached are the same for all three workloads.
NC-140 - No-Cache Peak, attained with 140 clients.
PC-140 - Page Cache Statistic, with 140 clients.
PC-280 - Page Cache Peak, attained with 280 clients.

Figure 4.11 : Performance of Page Cache for Shopping

NC-380 - No-Cache Peak, attained with 380 clients.
PC-380 - Page Cache Statistic, with 380 clients.
PC-400 - Page Cache Peak, attained with 400 clients.

Figure 4.12 : Performance of Page Cache for Ordering
<table>
<thead>
<tr>
<th>Cache Statistic</th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit Rate</td>
<td>0.863</td>
<td>0.787</td>
<td>0.754</td>
</tr>
<tr>
<td>Lookups per Interaction</td>
<td>0.951</td>
<td>0.805</td>
<td>0.501</td>
</tr>
<tr>
<td>Attempted Invalidations per minute</td>
<td>35.733</td>
<td>22.267</td>
<td>307.667</td>
</tr>
<tr>
<td>Actual Invalidations per minute</td>
<td>34.600</td>
<td>21.933</td>
<td>20.467</td>
</tr>
</tbody>
</table>

Table 4.4: Page Cache Statistics at Peak Performance for TPC-W Workloads

The actual invalidation rate is dependent on whether the results are cached or not when the invalidations are attempted - hence, the discrepancy between attempted-invalidations rate and the actual-invalidations rate.

To summarize the DCache’s performance, the cache is beneficial for all the three workloads. The extent of benefit is dependent on the extent of cache usage there is in the workloads. With the DCache also reducing the dynamic-content-generation overhead on the web server, it does not experience the performance loss for ordering (which has a higher load on the web server) as does the QCache.

4.3.4 Combining Cache Components

This section presents the results from combining all the three caching components — query cache, script cache and page cache. Figures 4.13, 4.14 and 4.15 give the performances for browsing, shopping and ordering, respectively. As before, NC denotes the base system in the graphs. AC denotes ‘all caches’ — the combined caching strategy that this thesis proposes. The comparisons are between the base system its peak (NC-80 for browsing, NC-140 for shopping, and NC-380 for ordering) and the all cache runs at the same loads and at the loads producing their peak
performances.

From figure 4.13, we see that combining the three caches improves both the throughput and response time very significantly. With 80 clients, the response time falls to under 4% of the base system's response time. Even at caching's peak throughput its response time is just 35% of the base system's. This is accompanied by throughput increase to a 8.7 times the peak throughput of the base system.

The significant improvements over the individual caching techniques (query cache in figure 4.3 and page cache in figure 4.10) indicates the synergistic effect between the caching components. Script cache complements query cache very nicely by addressing the web server bottleneck that arises when the query cache is in use. When DCache reduces both the web server's and database server's overheads for dynamic content it still retains the original bottleneck of the database component - once a query reaches the database system, it still has to be completely evaluated even if it produces cacheable results. The query cache by targeting the left over database overheads complements the DCache very well.

The improvements for shopping shown in figure 4.14 are similar to that for browsing - combining the cache components has a synergistic effect. The peak throughput increases to 4.4 times the base system's throughput. At the same load, response time reduces to just over 5% of the response time of the base system and at its peak, caching's response time is comparable to the base system's.

Ordering's performance also receives a significant boost compared to the case of caches in isolation - this is again brought about by the complementary roles of the three cache components. At the same load there is an increase in throughput by 22% with a reduction in response time of 90%. At caching's peak throughput the improvement is by a factor of 1.6 while seeing a 34% reduction in the response time.
NC-80 - No-Cache Peak, attained with 80 clients.
AC-80 - All Caches Statistic, with 80 clients.
AC-680 - All Caches Peak, attained with 680 clients.

Figure 4.13 : Performance of Caching Strategy for Browsing

NC-140 - No-Cache Peak, attained with 140 clients.
AC-140 - All Caches Statistic, with 140 clients.
AC-640 - All Caches Peak, attained with 640 clients.

Figure 4.14 : Performance of Caching Strategy for Shopping
Ordering's improvements are significantly lower because of the reduced usage of the data caching components - DCache and QCache.

With the current configuration I find that the peak performance is limited by the amount of memory on the web server. As the number of concurrent clients is increased beyond this point the web server starts paging reducing performance. The additional memory needs compared to the no-caching situation arises from two factors:

- The number of web server processes that are runnable at any time is significantly higher with caching because of the reduced computation time for each request and because of the increased number of clients that they are servicing. Thus, the memory needs of the server increase to service the increased throughput.

- The footprint of each process also increases with the addition of the DCache and Script Cache modules both of each run within the web server processes. The typical increase in footprint size observed was between 1-2 MB per process with a no-caching footprint of around 4MB. These numbers vary with the load on the server.

To summarize, combining the caching components increases performance for all the workloads. The components by addressing different sources of bottlenecks are quite complementary to each other thus boosting overall performance much greater than their individual ones. The extent of benefit for the workloads is again proportional to the extent which they can make use of cached results.
4.3.5 Storage Costs for Caching

The storage cost for each level of caching is quite low for the benefit produced. For the script cache, with the TPC-W workloads which use only 14 different content generation scripts, the storage space required is that for 14 compiled versions of these files which was just over 1MB. For the query cache, 11MB was the space required for the cache tables including the space for their indices. For DCache the maximum cache storage space required was around 50MB including the space for the in-memory hashtable.

For both query cache and DCache, the reduced storage requirements come from controlling the set of results that can be be cached. For query cache, the application-controlled cache design lets us choose the queries for which caching is to be performed - I use caching only for the queries where a cache hit would be beneficial and which would see a reasonable extent of re-use. With DCache, uncacheable pages (with
update queries) and pages with no re-use value can be marked as such with suitable headers generated during the response generation. Further, the DCache design allows us to make effective use of the cache slots by user-guided hashtable mapping that prioritizes/reserves slots for more expensive responses, thus, limiting the storage overheads. By using both, I ensure that caching is not done for responses without re-use and that there are always cache slots available for the results of expensive interactions.

4.4 Caching on Alternate Platform

As mentioned in the previous section, increased performance with caching requires sufficient memory on the web server. In this section, a comparison between the performance of the caching techniques on two different system configurations is presented. The first is the same one as used for the previous section - 2 1.33GhzZ AMD powered servers with 1.5GB memory on the web server and 768MB memory on the database server - this we refer to as the higher end (HE) system. The other platform it is compared to has 2 800Mhz AMD powered machines for its servers with 768MB memory on the web server and 256MB on the database server - this will be referred to as the lower end (LE) system.

First, I present the base system performance for both the configurations. Following that the performance for each of the caching techniques is examined, and finally the combined caching performance. As the base system performances are quite different, comparisons in the the latter sections are based on relative performances of the caching techniques and not their absolute performance.
4.4.1 Base System Performance

Figure 4.16 gives the relative peak performance for both the base configurations for each of the TPC-W workloads. Table 4.5 lists the number of concurrent clients at which the peak performance is attained for each system configuration and workload. It can be seen that the performance for the HE system configuration is considerably better than the LE one indicating the CPU-bound nature of the workloads.

Another interesting difference to note is the sharper drop in performance for browsing as compared to the other workloads under LE - this is more clearly visible as the relatively larger response time for its peak performance as compared to shopping in the LE system while having a marginally lower response time in the HE system. Given the database-bound nature of the performance for the browsing workload this indicates an even greater relative load on the database server in the LE system.
<table>
<thead>
<tr>
<th></th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>LE</td>
<td>40</td>
<td>50</td>
<td>200</td>
</tr>
<tr>
<td>HE</td>
<td>80</td>
<td>140</td>
<td>380</td>
</tr>
</tbody>
</table>

Table 4.5: Number of Concurrent Clients for Peak Performance on the Two System Configurations

### 4.4.2 Caching Performance

Figure 4.17 shows the relative performances of the Query Cache and the Page Cache for the two configurations. The two configurations exhibit significant difference in the improvements obtained by the caching techniques on them. The improvements for QCACHE are much higher on the LE system than on the HE system. Even ordering which saw a slight worsening in performance on the HE system, has an improvement with QCACHE on the LE system. This is because the differences in the load on the database server and web server is even more so on the LE system than on the HE system. The QCACHE system by eliminating this bottleneck is, thus, able to bring a bigger improvement on the LE system than on the HE system.

On the other hand, the Page Cache does much better relatively on the HE system than on the LE system. For each interaction, the page cache eliminates the dynamic content overheads from both the web server and database server, not providing any greater relief to the much more overloaded database server in LE. Thus, the system becomes bottlenecked once again at a higher throughput by the database performance sooner on LE than on the HE system. The HE system with a relatively more balanced set of servers (in terms of their load) is able to support a much higher growth in performance with the page cache before the database becomes
Figure 4.17: Relative Performances of Query Cache and Page Cache for the Two System Configurations

a bottleneck again.

In both platforms, while it is the database system that is the bottleneck the script cache’s role does not address that. Hence, the relative performances of the script cache over the base system is pretty much the same on both platforms - no significant change in performance.

Figure 4.18 presents the relative improvements in performance of the combined caching strategy on both platforms. It can be seen that there is considerable improvement on both platforms. In the HE system the improvements are a little smaller mainly because the system runs out of memory on the web server before any of the server CPUs can be fully utilized. On LE, the peaks are reached when the web server becomes fully utilized.

4.5 Changing Workload Characteristics

In this section, we look at the effect of some of the other characteristics of the workloads besides the interaction mix. First we look at the result for a smaller
Figure 4.18: Relative Performances of Combined Caching Strategy for the Two System Configurations

database size and then for a database with a larger number of items.

4.5.1 Database Size

The number of customers in the database, which is one of the two TPC-W scaling factors, affect the size of the database significantly. To get the performance for a database that would fit within memory the number of customers is set to 288K from 2.88 million, which I used for the earlier results. This results in a database size of 350MB which is smaller than the 768MB of memory we have on the database system. This comparison for the two database sizes would also show the effect of any special data caching techniques that might benefit database performance for TPC-W in the relative better performance the smaller database can get over the other. Table 4.6 lists the two database configurations we consider to evaluate the effect of database sizes. Figure 4.19 shows the base system throughputs without caching for both database sizes. As can be seen, their performances are comparable vindicating the CPU bound nature of the TPC-W workloads. This also underlines
<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Customers</th>
<th>Database Size</th>
<th>Database Server Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>2880000</td>
<td>3452 MB</td>
<td>768 MB</td>
</tr>
<tr>
<td>Small</td>
<td>288000</td>
<td>350 MB</td>
<td>768 MB</td>
</tr>
</tbody>
</table>

Table 4.6: Configuration for Evaluating Effect of Database Size on Caching

![Figure 4.19: Throughput for Different Database Sizes Without Caching](image)

Figure 4.19: Throughput for Different Database Sizes Without Caching

the relative marginalization of the contributions from potentially more efficient data caching techniques for these workloads at least on the system without the proposed caching techniques.

With caching, in general, the throughputs are limited by the amount of memory on the web server i.e. neither server is utilized to its peak capacity before the web server starts paging. Figure 4.20 shows the improvement in performance with caching for both the workloads. The smaller database size gets significantly lower improvement with caching.

The reason for this lower peak performance for the smaller database can be found in table 4.7 which shows the number of concurrent clients at which the peak through-
Figure 4.20: Improvements in Throughput for Different Database Sizes

<table>
<thead>
<tr>
<th></th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>680</td>
<td>640</td>
<td>560</td>
</tr>
<tr>
<td>Small</td>
<td>560</td>
<td>520</td>
<td>480</td>
</tr>
</tbody>
</table>

Table 4.7: Number of Concurrent Clients at Peak Performance for Different Database Sizes

Figure 4.21: Breakdown of Response Time for Different Database Sizes
put is attained for each workload under caching - the numbers are significantly lower for the smaller database size. This implies that as the client load is increased on the site the web server runs out of memory earlier for the smaller database size than for the regular one. Figure 4.21 that shows the breakdown in the response time components illustrates the reason for this behavior. It can be seen that for every corresponding configuration the fraction of time spent in the web server (Script plus Web Server) is greater for the smaller database size. With the smaller database size results are indeed returned from the database faster and in spending a longer time in the web server they occupy more resources, particularly memory. This results in their reaching the memory threshold earlier than the larger database size runs.

For the base system without caching, this effect is not observed as the database system is the dominant bottleneck for both database sizes and the this effect on the web server is marginalized.

4.5.2 Number of Items

In this section, we look at the effect of increasing the number of items in Item table. Table 4.8 lists the two configurations I consider - the Regular one which was used for all the other experiments with 10,000 items and the Large one which has 100,000 items. The effect of increasing the number of items on the database size is small, however, it results in a proportional increase in the storage size for images directory on the web server. The number of items can also play a role in the repetitiveness of requests to the server potentially affecting cache performance.

Figure 4.22 gives the throughputs of the base system without caching for both the configurations - it shows that base system peak performance is quite similar but marginally lower for the Large case. The small difference arises because of some of
<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Items</th>
<th>Database Size</th>
<th>Images on Web Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>10000</td>
<td>3452 MB</td>
<td>179 MB</td>
</tr>
<tr>
<td>Large</td>
<td>100000</td>
<td>3508 MB</td>
<td>1804 MB</td>
</tr>
</tbody>
</table>

Table 4.8: Configuration for Evaluating Effect of Number of Items on Caching

![Varying Number of Items - Throughputs without Caching](image)

Figure 4.22: Base System Throughput for Different Number of Items
the search queries needing to do additional work proportional to the increase in the number of items. These queries being more dominant in browsing and shopping, their effects are also more noticeable there.

Figure 4.23 gives the normalized improvements in peak throughput with caching for the Regular and Large configurations. It also gives the figures for the throughput improvements for the Regular configuration at the load at which Large gets its peak throughputs - these are denoted as Regular-Large. All peak performances are limited by the memory on the web server. The peak performances for Large are clearly lower than Regular's. But they are quite comparable to Regular-Large's indicating that the difference in the repetitiveness of the requests arising from the difference in the number of items did not alter performance significantly.

The reason for the difference in performance between Large and Regular comes from the difference in the number of different image files served from the web server. They are ten times more for Large resulting in a much larger utilization of the file
Table 4.9: Number of Concurrent Clients at Peak Performance for Different Number of Items

<table>
<thead>
<tr>
<th></th>
<th>Browsing</th>
<th>Shopping</th>
<th>Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>680</td>
<td>640</td>
<td>560</td>
</tr>
<tr>
<td>Large</td>
<td>360</td>
<td>360</td>
<td>400</td>
</tr>
</tbody>
</table>

cache on the web server by the image files. This adds to the memory pressure on the web server resulting in the performance peaking at a lower number of concurrent clients (show in table 4.9) and a correspondingly lower utilization of the web server.

4.6 Summary

The main results from the evaluation of the caching strategy are

- The database component is the main bottleneck for the base system without caching. The query cache improves performance significantly for browsing and shopping. However, its overheads bring about a small reduction in performance for ordering whose database overheads sees little reduction from caching, while its web server overheads increase with query caching. The script cache does little to improve performance for the base system as its benefit is in reducing overheads on the web server while the system's main bottleneck is on the database server. The page cache brings the best improvement in performance because of complete elimination of dynamic content overheads on cache hits.

- Combining all the caches increases performance significantly over any single cache performance showing the synergistic behavior of the caches. At the
peak performance for caching, the combined cache throughputs are 8.72, 4.39 and 1.58 times the no-cache throughputs for browsing, shopping, and ordering respectively. At these increased throughputs, browsing and ordering see a reduction in response time of 65% and 34%, respectively, with shopping's showing little change. The reductions in the response times at the same load as the no-cache system's peak throughput are 97%, 95%, and 90% for browsing, shopping, and ordering, respectively. The improvements vary due to the difference in the cache usage characteristics of the three workloads. The peak performance is limited by the memory on the web server for the system with all the caches indicating the need for additional memory on the web server for seeing the best performance from caching. The storage requirements for these benefits are kept quite small because of the basic design of the caches and by utilizing site-specific optimizations for both the query cache and page cache.

- When the cache performances for two different machine configurations are compared, the lower end (LE) configuration sees a difference in relative behavior of the query cache and page cache compared to the higher end (HE) configuration. LE is more critically bottlenecked at the database system than HE, and the query cache that directly targets this bottleneck achieves greater improvement for it than the page cache. While for HE, it is the page cache that is the better performer. The combined cache performance is comparable for both with the LE system seeing slightly higher improvements - this is because it reaches the memory ceiling on the web server at a later stage than the HE system which has faster processing capabilities.

- When the cache performance for two databases of different sizes - one smaller than the amount of database memory (Small) and the other larger (Regular)
are compared we find that Small obtains lesser benefits from caching than Regular. This is because, Small, has a larger proportion of its work on the web server than Regular. As a result of this, it reaches the memory ceiling on the web server sooner i.e. at a lower number of concurrent clients and correspondingly lower utilizations of the servers. This results in lower improvements in peak performance for Small.

- For two databases with different values for the number of items scale factor of TPC-W - referred to as Regular for 10,000 items and Large for 100,000 items - there is significant difference in cache performance. Here again, the Large configuration reaches the memory ceiling on the web server earlier resulting in lower performance for it. The reason for it is that the Large configuration serves ten times larger number of files off the web server resulting in it using up the memory on the web server sooner. This results in the peak performance for Large being attained at lower number of concurrent clients to the site and at correspondingly lower utilizations of the servers. A comparison of Large's performance with Regular's at the same number of concurrent clients shows them to be similar indicating that Large could obtain similar improvements as Regular without the memory ceiling, i.e., with additional memory.
Chapter 5
Conclusions and Future Work

5.1 Summary of Contributions

Database-driven web sites are the primary mode of dynamic content delivery that enables much of the features of modern web sites. I address the serious performance problem with current infrastructure for dynamic content delivery. My solution is to adopt a dynamic content caching strategy that is enabled with multiple, independent caching components, each one focusing on optimizing performance at its own tier. My key contributions

- I proposed a novel multi-tier caching strategy that incorporates multiple caching components that can be deployed transparently over the existing multi-layered architecture for dynamic content delivery. Each caching component can eliminate specific bottlenecks in the system providing a unique contribution. In addition, the components can combine well to obtain the best performance for the overall system. The improved performance can be obtained without requiring a change in the content generation tools or methodologies while utilizing user directives independent from the content generation approach.

- I provide a detailed quantitative evaluation of the benefits of dynamic content caching for the performance of database-driven web sites. The experiments with e-commerce workloads indicate that the database system could be a sig-
significant bottleneck requiring a caching component at the database layer to address this. They further demonstrate the effectiveness of a multi-tier caching strategy and the synergy between its different caching components. This synergy enables an overall improvement in performance significantly greater than any of their individual contributions and performance almost an order of magnitude greater than the non-caching performance for some of the workloads.

5.2 Future Work

My work in dynamic content caching can be extended in many ways. The following sections give a brief overview of my thoughts on them.

5.2.1 Other Dynamic Content Workloads

While TPC-W is representative of retail e-commerce workloads which forms an important subset of dynamic content workloads, there are other kinds of web site workloads for which caching can be beneficial. Notable and popular among them are bulletin boards and the related messaging services, online news services, portals for a variety of subjects and interests, and information repositories such as online digital libraries. We have begun work on bulletin-board workloads with an emulation of a slashdot-like site. Slashdot [46] is one of the most popular bulletin board sites that caters to discussions on a variety of topics largely related to the computer technologies and industry. The content generation and management code for the site has been made available for public use at the Slashcode [47] site which we have utilized to guide our analysis of bulletin board workloads. Preliminary findings indicate a site with more heavily loaded web servers than database servers. This provides a very different setting for the caching architecture from the TPC-W workloads. We
believe the multi-tier aspect of our solution with components to tackle each potential bottleneck source could be quite effective in improving the performance for the new workloads too. We plan to investigate this in greater detail.

5.2.2 Cache Implementation with Automatic Dependency Detection

In the work presented here, I rely primarily on user-input to guide the consistency actions for cached objects - through directives for the DCache system and application code changes for the QCache system. I plan to develop an automatic dependency detection system that can identify dependencies between the update and read request streams to the database system. Once developed this could eliminate or reduce the need for user intervention in cache management.

I believe, the first step towards automatic dependency detection is to partition queries into sub-classes which require different sets of information for dependency detection. For some the dependencies can be completely inferred from the conflicting queries alone, for others only a conservative decision can be reached from the queries - more accurate determination could require access to the database metadata (table and key definitions), data in the cached results, or even related data from other tables (e.g. in the case of joins). The mechanisms to obtain this additional information are determined by the transparency and location of the caches. A transparent cache would have to use conventional querying mechanisms to obtain this information while a cache integrated with the database system can use specially designed procedures (including additional cache metadata support from the database system) to obtain this information. The location of the cache would determine the communication method used by it to retrieve this information.

The performance trade-offs for conservative decisions versus decisions made with
complete information also need to be explored as the costs for obtaining the complete information could exceed the benefits from retaining the result in cache. The extent of support both in the classes of queries that can be cached and in the mechanisms to enable caching for them could also be influenced by the nature of the workloads. Additional issues involve the support of variable consistency levels for different classes of results which would also play a role in determining the mechanisms needed to provide them.

I plan to examine the above issues in detail while developing caches with automatic dependency detection.

5.2.3 Caching Solutions for Server Farms

I find that many interesting issues that come up when extending the caching solutions to a cluster of servers. Chief among them are

- Maintenance of consistency among potentially multiple caching servers - using centralized versus distributed cache coherence solutions.

- Distribution of resources - how much to allocate to each function such as web servers, caching servers, and auxiliary servers such as the database server. A related issue is a system design where server functionality can be altered with minimal or no down time to match the needs.

- Effective marriage between load balancing and caching - related issues have been addressed for static content [48], however, the nature and magnitude of the computational load for dynamic content brings forth many new issues to be addressed.
Considering these from the context of the proposed multi-tier caching strategy brings about additional issues to investigate. I expect to be actively engaged in analyzing and solving these questions.

5.2.4 Dynamic Content Caching at the Caches on the Network

The Active Cache [38] framework provides a method to keep dynamic content caches consistent at the proxies. It, however, requires that proxies execute server-provided applets on them. This raises some performance and security issues which need to be addressed before the method can be adopted. The Edge Side Includes (ESI) initiative [49] attempts to provide a simple markup language for dynamic assembly of web page components at caching servers on the edges of the internet (like those of content distribution networks or CDNs). The approach is based on caching servers cooperating with the web sites to accept expiration and invalidation messages with ESI for cached content.

Both these approaches can be used to augment and extend my work's caching solutions to the network. I plan to investigate the choice of incorporating these approaches or developing new solutions from scratch to extend the web site caching technology developed in this thesis to the network.
Bibliography


1996.


ence on Distributed Computing and Systems (ICDCS), May 1999.


Appendix A

TPC-W Queries

This Appendix gives all pertinent information on the database tables and queries in the TPC-W workloads. There are 10 main tables in TPC-W: Item, Author, Customer, Address, Country, Orders, Order_line, CC_Xacts, Shopping_cart, and Shopping_cart_line.

The queries are listed under the interaction(s) in which they occur. The query under Promotional Processing is not in a separate interaction, rather, it is common to five interactions - Home, BestSellers, NewProducts, SearchRequest, and SearchResults - and, is listed first. Before each query the tables involved in it and the indices used for the regular table sizes are also given.

A.1 Promotional Processing

Tables: item
Index: item.i_id (primary)

```
SELECT J.i_id, J.i_thumbnail
FROM item I, item J
WHERE (I.i_related1 = J.i_id or I.i_related2 = J.i_id or I.i_related3 = J.i_id or I.i_related4 = J.i_id or I.i_related5 = J.i_id) and I.i_id = '<input>';  
```

A.2 Home

Tables: customer
Index: PRIMARY (c_id)

```
SELECT c.fname, c.lname
```
FROM customer
WHERE c.id = '<input>';

A.3 NewProducts

Tables: item, author
Index: item.i.subject.i.a.id.i.id.i.title (where used, using filesort), PRIMARY (a.id)

SELECT i.id, i.title, a.fname, a.lname, i.subject
FROM item, author
WHERE item.i.a.id = author.a.id AND item.i.subject = '<input>'
ORDER BY item.i.pub.date DESC, item.i.title
LIMIT 0,50;

A.4 BestSellers

Tables: orders
Index: orders.o.date.o.id

CREATE TEMPORARY TABLE tmp.bs TYPE=HEAP
SELECT o.id
FROM orders
ORDER BY o.date DESC
LIMIT 3333;

Tables: tmp.bs, order.line, item, author
Index: PRIMARY ((ol.o.id, ol.ol.id)), PRIMARY (i.id), PRIMARY (a.id)

SELECT i.id, i.title, a.fname, a.lname, SUM(ol.qty) AS val
FROM tmp.bs, order.line, item, author
WHERE order.line.ol.o.id=tmp.bs.o.id and item.i.id = order.line.ol.i.id
AND
item.i.subject = '<input>' AND item.i.a.id = author.a.id
GROUP BY i.id
ORDER BY val DESC
LIMIT 0,50;
Tables: tmp bs DROP TABLE tmp bs;

A.5 ProductDetail

Tables: item, author
Index: PRIMARY (i_id), PRIMARY (a_id)

SELECT i.*, a.a_id, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i_a_id = a.a_id AND i.id = <input>;

A.6 SearchResult

A.6.1 By Author

Tables: item, author
Index: author.a_lname_a_id.a_fname, item.i_a_id.

SELECT i.id, i.i_title, a.a_fname, a.a_lname
FROM item i, author a
WHERE a.a_lname LIKE <input> AND i.i_a_id = a.a_id
ORDER BY i.i_title
LIMIT 0,50;

A.6.2 By Title

Tables: item, author
Index: author.a_lname_a_id.a_fname, item.i_a_id

SELECT i.id, i.i_title, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i_a_id = a.a_id AND i.i_title LIKE <input>
ORDER BY i.i_title
LIMIT 0,50;

A.6.3 By Subject

Tables: item, author
Index: item.i.subject (where used; using filesort), PRIMARY (a.id)

SELECT item.i.id, item.i.title, author.a.fname, author.a.lname
FROM item, author
WHERE item.i.a_id = author.a_id AND item.i.subject = '<input>
ORDER BY item.i.title
LIMIT 0,50;

A.7 OrderDisplay

Table: customer
Index: customer.c.uname (where used)

SELECT c.id, cpasswd
FROM customer
WHERE c.uname = '<input'>;

Table: orders
Index: Comment - select tables optimized away - because of running optimze on orders (indices are sorted) and the using index orders.o.c.id.o.id the max o.id for every o.c.id is available in the index file. SELECT max(o.id)

FROM orders
WHERE orders.o.c.id = '<input'>;

Tables: customer, orders, cc.xacts, address (2), country (2)
Index: PRIMARY (o.id), PRIMARY (cx.o.id), PRIMARY (addr.id), PRIMARY (co.id), PRIMARY (addr.id), PRIMARY (co.id), PRIMARY (c.id). SELECT
orders.o.id, orders.o.date, orders.o.sub_total, orders.o.tax,
orders.o.total, orders.o.ship.type, orders.o.ship.date,
orders.o.status, customer.c.uname, customer.c.passwd,
customer.c.fname, customer.c.lname, customer.c.phone,
customer.c.email, cc.xacts.cx.type, ship.addr.street1 AS
ship.addr.street1, ship.addr.street2 AS ship.addr.street2,
ship.addr.state AS ship.addr.state, ship.addr.zip AS ship.addr.zip,
ship.co.co.name AS ship.co.name, bill.addr.street1 AS
bill_addr.street1, bill_addr.street2 AS bill_addr.street2,
bill_addr.state AS bill_addr.state, bill_addr.zip AS bill_addr.zip,
bill.co.co_name AS bill.co.name
FROM customer, orders, cc.xacts, address ship, country ship.co,
address bill, country bill.co
WHERE orders.o.id = <input> AND cx.o.id = orders.o.id AND
customer.c.id = orders.o.c.id AND orders.o.bill_addr.id = bill.addr.id
AND bill.addr.co.id = bill.co.co.id AND orders.o.ship.addr.id =
ship.addr.id AND ship.addr.co.id = ship.co.co.id

Tables: order_line, item
Index: PRIMARY ((ol.o.id, ol.id)), PRIMARY (i.id)

SELECT ol.i.id, ol.qty, ol.discount, ol.comments, i.title, i.publisher,
i.cost FROM order_line, item
WHERE ol.o.id = <input> AND ol.i.id = i.id;

A.8 ShoppingCart

Tables: shopping_cart

INSERT INTO shopping_cart (sc.id, sc.time) VALUES (<input>, NOW())

Tables: shopping_cart.line
Index: PRIMARY ((scl.sc.id, scl.i.id))

SELECT scl.qty
FROM shopping_cart.line
WHERE scl.sc.id = <input> AND scl.i.id = <input>;

Tables: shopping_cart.line
Index: PRIMARY ((scl.sc.id, scl.i.id))

UPDATE shopping_cart.line
SET scl.qty = <input> WHERE scl.sc.id = <input> AND scl.i.id = <input>;

Tables: shopping_cart.line

INSERT into shopping_cart.line (scl.sc.id, scl.qty, scl.i.id) VALUES
(<input>, <input>, <input>);

Tables: shopping_cart
Index: PRIMARY ((scl_sc_id, scl_i_id))

DELETE FROM shopping_cart
WHERE scl_sc_id = <input> AND scl_i_id = <input>;

Tables: shopping_cart
Index: PRIMARY ((scl_sc_id, scl_i_id))

SELECT COUNT(*)
FROM shopping_cart
WHERE scl_sc_id = <input>;

Tables: shopping_cart, item
Index: PRIMARY ((scl_sc_id, scl_i_id)), PRIMARY (i.id)

SELECT *
FROM shopping_cart, item
WHERE scl_i_id = item.i_id AND scl_sc_id = <input>;

Tables: shopping_cart
Index: PRIMARY (sc.id)

UPDATE shopping_cart
SET sc_time = NOW()
WHERE sc_id = <input>;

A.9 BuyRequest

Tables: customer, address, country
Index: customer.c_uname, PRIMARY (addr_id), PRIMARY (co_id)

SELECT c_id, c.uname, c.passwd, c.fname, c.lname, c.phone, c.email, c.since, c.last_login, c.login, c.expiration, c.discount, c.balance, c.ytd_pmt, c.birthday, address.*, co.name
FROM customer, address, country
WHERE customer.c_addr_id = address.addr_id AND address.addr_co_id = country.co_id AND customer.c_uname = <input>;
Tables: customer
Index: PRIMARY (c_id)

UPDATE customer
SET c_login = NOW(), c_expiration = DATE_ADD(NOW(), INTERVAL 2 HOUR)
WHERE c_id = '<input>';

Tables: country
Index: PRIMARY (co_id)

SELECT co_id
FROM country
WHERE co_name = '<input>';

Tables: address
Index: addr_zip

SELECT addr_id
FROM address
WHERE addr_street1 = '<input>' AND addr_street2 = '<input>' AND
addr_city = '<input>' AND addr_state = '<input>' AND addr_zip =
'<input>' AND addr_co_id = '<input>';
SELECT max(c.id)
FROM customer;

Tables: customer

INSERT INTO customer (c.id, c.uname, c.passwd, c.fname, c.lname,
c.addr_id, c.phone, c.email, c.since, c.last_login, c.login,
c.expiration, c.discount, c.balance, c.ytd_pmt, c.birthdate, c.data)
VALUES (< input >, < input >, < input >, < input >, < input >,
< input >, < input >, < input >, CURDATE(), CURDATE(), NOW(),
DATE.ADD(NOW(), INTERVAL 7200 SECOND), < input >, < input >,
< input >, < input >, < input >) Tables: shopping_cart_line

Index: PRIMARY ((scl.sc_id, scl.i_id)), PRIMARY (i.id)

SELECT *
FROM shopping_cart_line, item
WHERE scl.i_id = item.i_id AND scl.sc_id = < input >;

A.10 BuyConfirm

Tables: customer
Index: PRIMARY (c.id)

SELECT c.discount
FROM customer
WHERE customer.c.id = < input >;

Tables: shopping_cart_line
Index: PRIMARY ((scl.sc_id, scl.i_id)), PRIMARY (i.id)

SELECT *
FROM shopping_cart_line, item
WHERE scl.i_id = item.i_id AND scl.sc_id = < input >;

Tables: customer
Index: PRIMARY (c.id)
SELECT c.addr_id
FROM customer
WHERE customer.c_id = '<input>';

Tables: orders
Index: Comment - Select tables optimized away

SELECT max(o_id)
FROM orders:

Tables: orders

INSERT into orders (o_id, o.c_id, o.date, o.sub_total, o.tax, o.total, o.ship_type, o.ship_date, o.bill_addr_id, o.ship_addr_id, o.status) VALUES ('<input>', '<input>', NOW(), '<input>', 8.25, '<input>', '<input>', DATE_ADD(CURRENT_DATE(), INTERVAL '<input>' DAY), '<input>', '<input>', 'Pending');

Tables: order_line

INSERT into order_line (ol.id, ol_o.id, ol_i.id, ol_qty, ol_discount, ol_comments)
VALUES ('<input>', '<input>', '<input>', '<input>', '<input>', '<input>', '<input>') Tables: item

Index: PRIMARY (i.id)

UPDATE item
SET i.stock = i.stock-'<input>'
WHERE i.id = '<input>' and i.stock-'<input>' >= 10;

Tables: item
Index: PRIMARY (i.id)

UPDATE item
SET i.stock = i.stock-'<input>'+21
WHERE i.id = '<input>';
Tables:  address
Index:  PRIMARY (addr_id)

SELECT addr.co_id
FROM address
WHERE addr_id = <input>;

Tables:  cc.xacts

INSERT into cc.xacts (cx.o_id, cx.type, cx.num, cx.name, cx.expiry,
                      cx.xact.amt, cx.xact.date, cx.co_id)
VALUES (<input>, <input>, <input>, <input>, <input>,
        <input>, CURRENT_DATE(), <input>);

Tables:  shopping_cart.line
Index:  PRIMARY ((scl.sc_id, scl.i_id))

DELETE
FROM shopping_cart.line
WHERE scl.sc_id = <input>;

A.11  AdminRequest

Tables:  item, author
Index:  PRIMARY (i.id), PRIMARY (a.id)

SELECT i.*, a.a.id, a.a.fname, a.a.lname
FROM item i, author a
WHERE i.i.a_id = a.a.id AND i.i.id = <input>;

A.12  AdminConfirm

Tables:  item, author
Index:  PRIMARY (i.id), PRIMARY (a.id)

SELECT i.*, a.a.id, a.a.fname, a.a.lname
FROM item i, author a
WHERE i.i.a.id = a.a.id AND i.i.id = <input>;
CREATE TEMPORARY TABLE tmp_admin TYPE=HEAP
SELECT o.id
FROM orders
ORDER BY o.date DESC
LIMIT 10000;

SELECT ol2.ol.i.id, SUM(ol2.ol.QTY) AS sum.ol
FROM order_line ol, order_line ol2, tmp_admin t
WHERE ol.ol.o.id=t.o.id AND ol.ol.i.id= <input> AND ol2.ol.o.id =
t.o.id AND ol2.ol.i.id <> <input>
GROUP BY ol2.ol.i.id
ORDER BY sum.ol DESC
LIMIT 5;

DROP TABLE tmp_admin;

UPDATE item
SET I.COST= <input>, I.THUMBNAIL=<input>, I.IMAGE=<input>,
I.PUB_DATE = <input>, I.RELATED1 = <input>, I.RELATED2 = <input>,
I.RELATED3 = <input>, I.RELATED4 = <input>, I.RELATED5 = <input>
WHERE I.ID = <input>
A.13 CustomerRegistration

Tables: customer
Index: PRIMARY (c_id)

SELECT c.uname
FROM customer
WHERE c.id = <input>;}
Appendix B

Query Modification for Query Cache

This Appendix lists all the modifications to the queries in the application scripts for implementing query caching. In each instance, the original query is listed first and then the queries required for query caching which replace the original query. In the cases where queries are required to invalidate entries in the cache tables, there is no matching original query and, hence, non listed.

B.1 Promotional Processing

Original query:

```
SELECT J.i_id, J.i_thumbnail
FROM item I, item J
WHERE (I.i_related1 = J.i_id or I.i_related2 = J.i_id or I.i_related3
  = J.i_id or I.i_related4 = J.i_id or I.i_related5 = J.i_id) and I.i_id
  = '<input>'; For Query Cache:
```

```
SELECT i_related, i_related_thumbnail
FROM getRelated_ct
WHERE i_id = '<input>);
```

If the result from the above query is empty then perform:

```
INSERT INTO getRelated_ct SELECT J.i_id, J.i_thumbnail, '<input>
FROM item I, item J
WHERE (I.i_related1 = J.i_id or I.i_related2 = J.i_id or I.i_related3
  = J.i_id
or I.i_related4 = J.i_id or I.i_related5 = J.i_id) and I.i_id = 
  '<input>' );
```
SELECT i.related, i.related_thumbnail
FROM getRelated.ct
WHERE i.id = '<input>';

B.2 NewProducts

Original Query:

SELECT i.id, i.title, a.fname, a.lname, i.subject
FROM item, author
WHERE item.i_a_id = author.a_id AND item.i_subject = '<input>
ORDER BY item.i_pub_date DESC, item.i_title
LIMIT 0,50;

Query Cache:

SELECT *
FROM getNewProducts.ct
WHERE i.subject = '<input>';

If the result from the above query is empty then perform:

INSERT INTO getNewProducts.ct
SELECT i.id, i.title, a.fname, a.lname, i.subject
FROM item, author
WHERE item.i_a_id = author.a_id AND item.i_subject = '<input>
ORDER BY item.i_pub_date DESC, item.i_title
LIMIT 0,50;

SELECT *
FROM getNewProducts.ct
WHERE i.subject = '<input>';

B.3 BestSellers

Original Query:

CREATE TEMPORARY TABLE tmp_bs
TYPE=HEAP
SELECT o.id
FROM orders
ORDER BY o.date DESC
LIMIT 3333;

SELECT i.id, i.title, a.fname, a.lname, SUM(ol.qty) AS val
FROM tmp.bs, order_line, item, author
WHERE order_line.ol.o.id=tmp.bs.o.id and item.i.id = order_line.ol.i.id
AND
item.i.subject = '<input>' AND item.i.a.id = author.a.id
GROUP BY i.id
ORDER BY val DESC
LIMIT 0,50;

DROP TABLE tmp.bs;

Query Cache:

SELECT * from getBestSellers_ct where i.subject=<input>

If the result from the above query is empty then perform:

CREATE TEMPORARY TABLE tmp_bs
TYPE=HEAP
SELECT o.id
FROM orders
ORDER BY o.date DESC
LIMIT 3333;

INSERT INTO getBestSellers_ct
SELECT i.id, i.title, a.fname, a.lname, SUM(ol.qty) AS val, '<input>'
FROM tmp.bs, order_line, item, author
WHERE order_line.ol.o.id = tmp.bs.o.id and item.i.id = order_line.ol.i.id
AND item.i.subject = '<input>' AND item.i.a.id = author.a.id
GROUP BY i.id
ORDER BY val DESC
LIMIT 0,50;

SELECT * from getBestSellers_ct where i.subject=<input>
B.4 ProductDetail

Original Query:

SELECT i.*, a.a_id, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i_a_id = a.a_id AND i.i_id = <input>;

Query Cache:

SELECT *
FROM getBook_ct
WHERE i_id = <input>;

If the result from the above query is empty then perform:

INSERT INTO getBook_ct
SELECT i.*, a.a_id, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i_a_id = a.a_id AND i.i_id = <input>;

SELECT *
FROM getBook_ct
WHERE i_id = <input>;

B.5 SearchResult

B.5.1 By Author

Original Query:

SELECT i.i_id, i.i_title, a.a_fname, a.a_lname
FROM item i, author a
WHERE a.a_lname LIKE <value>% AND i.i_a_id = a.a_id
ORDER BY i.i_title
LIMIT 0,50;

Query Cache:

SELECT *
FROM doAuthorSearch_ct
WHERE search_key = '<input>);

If the result from the above query is empty then perform:

INSERT INTO doAuthorSearch_ct
SELECT i.i.id, i.i.title, a.a.fname, a.a.lname, '<input>
FROM item i, author a
WHERE a.a.lname LIKE '<%value%>' AND i.i.a.id = a.a.id
ORDER BY i.i.title
LIMIT 0,50;

SELECT *
FROM doAuthorSearch_ct
WHERE search_key = '<input>;

B.5.2 By Title

Original Query:

SELECT i.i.id, i.i.title, a.a.fname, a.a.lname
FROM item i, author a
WHERE i.i.a.id = a.a.id AND i.i.title LIKE '<value%>
ORDER BY i.i.title
LIMIT 0,50;

Query Cache:

SELECT *
FROM doTitleSearch_ct
WHERE search_key = '<input>;

If the result from the above query is empty then perform:

INSERT INTO doTitleSearch_ct
SELECT i.i.id, i.i.title, a.a.fname, a.a.lname, '<input>
FROM item i, author a
WHERE i.i.a.id = a.a.id AND i.i.title LIKE '<value%>
ORDER BY i.i.title
LIMIT 0,50;

SELECT *
FROM doTitleSearch_ct
WHERE search_key = '<input>';

B.5.3 By Subject

Original Query:

SELECT item.i.id, item.i.title, author.a.fname, author.a.lname
FROM item, author
WHERE item.i.a.id = author.a_id AND item.i.subject = '<input>'
ORDER BY item.i.title
LIMIT 0,50;

Query Cache:

SELECT *
FROM doSubjectSearch_ct
WHERE i.subject = '<input>';

If the result from the above query is empty then perform:

INSERT INTO doSubjectSearch_ct
SELECT i.i.id, i.i.title, a.a.fname, a.a.lname, i.i.subject
FROM item i, author a
WHERE item.i.a.id = author.a_id AND item.i.subject = '<input>'
ORDER BY item.i.title
LIMIT 0,50;

SELECT *
FROM doSubjectSearch_ct
WHERE i.subject = '<input>';
B.7 AdminRequest

Original Query:

SELECT i.*, a.a_id, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i.a_id = a.a_id AND i.i.id = <input>;

Query Cache:

SELECT *
FROM getBook.ct
WHERE i.id = <input>;

If the result from the above query is empty then perform:

INSERT INTO getBook.ct
SELECT i.*, a.a_id, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i.a_id = a.a_id AND i.i.id = <input>;

SELECT *
FROM getBook.ct
WHERE i.id = <input>;

B.8 AdminConfirm

Original Query:

SELECT i.*, a.a_id, a.a_fname, a.a_lname
FROM item i, author a
WHERE i.i.a_id = a.a_id AND i.i.id = <input>;

Query Cache:

SELECT *
FROM getBook.ct
WHERE i.id = <input>;

If the result from the above query is empty then perform:

INSERT INTO getBook.ct
SELECT *, a.a_id, a.fname, a.lname
FROM item i, author a
WHERE i.i_a_id = a.a_id AND i.i_id = <input>;

SELECT *
FROM getBook.ct
WHERE i_id = <input>;

Query Cache (invalidate):

DELETE
FROM getBook.ct
WHERE i_id = <input>;

DELETE
FROM getNewProducts.ct
WHERE i_subject = <input>;

DELETE
FROM getRelated.ct
WHERE i_id = <input>;