Putting the "Scalability" into Database Scalability Services

Charles Garrod

CMU-CS-08-150

August 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Bruce M. Maggs, Chair
David Andersen
Anthony Tomasic
Christopher Olston (Yahoo! Research)
Mike Dahlin (University of Texas at Austin)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers CCR-0205544 and CNS-0435382. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Web Applications, Scalability, Publish / Subscribe, View Invalidation, View Materialization, Database Query Result Caching.

Abstract

Applications deployed on the Internet are immediately accessible to a vast population of potential users, and as a result they tend to experience unpredictable and widely fluctuating demand. System administrators currently face a provisioning dilemma to address this demand: whether to (1) waste money by heavily overprovisioning systems, or (2) risk loss of availability during times of high demand. This problem is largely solved for static Web content, but existing approaches do not apply well to dynamic content produced by data-intensive Web applications for which a central database server limits scalability.

To address this problem, we design and build a Database Scalability Service (DBSS), which can offer scalability to data-intensive Web applications as a third-party service much like Content Delivery Networks currently scale static Web content. The key challenge in building a DBSS is to enable the DBSS to off-load database requests from a content provider's central database server while ensuring that the DBSS uses up-to-date, consistent data as the database is updated. In addition, a DBSS faces the additional problems of maintaining high quality of service for each content provider as well as guaranteeing the privacy of a content provider's data.

In this thesis, we focus on the scalability-related aspects of a DBSS. We design and evaluate a DBSS, called Ferdinand, that uses a multi-tiered caching architecture, with a local database cache at each Ferdinand server and a shared, collaborative cache distributed among Ferdinand's nodes. Ferdinand maintains the consistency of the database caches using a fully distributed publish / subscribe system, notifying each cache of database updates without placing additional administrative load on the central database server. Our primary technique to efficiently maintain cache consistency is to specialize the publish / subscribe

system for each Web application, using an offline analysis of the Web application and its database requests. We use compiler-like techniques to advance the state-of-the-art in this offline analysis, allowing us to better understand how a Web application is affected by updates to its database. Overall, we show that our multi-tiered cache design and scalable consistency management are both critical at maximizing Ferdinand's scalability, and that the Ferdinand DBSS can scale the throughput of data-intensive Web applications by more than a factor of 10.

Acknowledgments

Any student will tell you that their dissertation is not just a product of their own efforts, but is the result of many contributions from a large number of people. For this work I had two great advisors, Bruce Maggs and Chris Olston. Both made large technical contributions to my dissertation: Bruce mostly to the caching work and Chris to managing data consistency. What I truly appreciate, however, is how they shaped me professionally. Bruce has the rare ability of teaching without calling attention to the student's current deficiencies, a skill I hope to attain. Combined with his great patience, he is a true proponent of the principle that if you give someone a chance and a reason to succeed, they will. Chris's greatest influence was adding "impact" to my vocabulary.

Dave Andersen, Mike Dahlin, and Anthony Tomasic all provided great advice and feedback at various points in this project. I also have the pleasure of seeing Dave and Anthony outside the scope of the project on a regular basis, and consider them both to be great friends.

The Ferdinand project grew from earlier work by Amit Manjhi, who focused on the privacy- and quality of service-related aspects of Database Scalability Services. Amit's earlier projects provided much of the framework for the design and evaluation of Ferdinand, and for that I owe him a great debt. Natassa Ailimaki, Phil Gibbons, and Todd Mowry all made significant technical contributions and gave important feedback on my research results. The feedback of Mukesh Agrawal, Michael Ashley-Rollman, Chris Colohan, Michelle Goodstein, Katrina Ligett, David McWherter, Shafeeq Sinnamohideen, and Kami Vaniea all improved the writing and presentation of the work.

Finally, the following people did not directly contribute to the Ferdinand project but had a large positive impact on my personal and professional life along the way: Steve and TC Altus, Manuel Blum,

Joanna Bresee, Susan and Jim Bresee, Matt Clark, Bob Harper, Benoît Hudson, Mike Erlinger, Jason Flinn, Nick Jong, Geoff Kuenning, Ran Libeskind-Hadas, Lara Mercurio, Adam Meyerson, Ellie and Nathan Ratliff, Rob Reeder, Megan Thorsen, and of course(!) my parents Curt and Vicki Garrod, my sister and her husband Celeste and Adam, and my grandparents Charles and Joyce Garrod.

Thank you all.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction to a Database Scalability Service (DBSS)

Applications deployed on the Internet are immediately accessible to a vast population of potential users. As a result, they tend to experience unpredictable and widely fluctuating degrees of load, especially due to events such as breaking news (e.g., 9/11), sudden popularity spikes (e.g., the "slashdot effect"), or denial-of-service (DoS) attacks. System administrators currently face a provisioning dilemma: whether to (1) waste money by heavily overprovisioning systems, or (2) risk loss of availability during times of high demand. In many situations both of these alternatives can be expensive and are undesirable.

This problem is largely addressed for static content (e.g., fixed Web pages, images, and video) by Content Delivery Network (CDN) technology [2, 24, 61, 74], which offers on-demand scalability as a plug-in service. Since most static content changes infrequently, a CDN can efficiently cache the content on many servers in a large distributed infrastructure, offloading work from the central content server and offering seemingly unlimited scalability. By sharing the distributed infrastructure among many content providers, the CDN can absorb load spikes that might occur for any single content provider while still being cost effective, charging each content provider on a per-usage basis.

The Web, however, is becoming increasingly more dynamic, with the content generated by programs executed at the time of each Web request rather than existing in a static file that is easily distributed. Dynamic Web applications allow the content to be customized based on factors such as user preferences,

previous content the user has viewed, or even extrinsic information such as the physical location of the user. Dynamic technologies such as Adobe Flash[1] and Asynchronous JavaScript And XML (AJAX) additionally allow a Web page to be constructed from multiple asynchronous Web requests, enabling a rich interactive experience that static content cannot provide. Since dynamic content is generated by programs, might depend on data not contained in the user's request, and might depend on data that changes frequently, current CDN technology does not adequately support dynamic content.

System administrators face the same provisioning problem for dynamic content that they do for static content: they must choose between overprovisioning system infrastructure for normal system loads or risk being unable to serve dynamic content during periods of high demand.

In this thesis, we show that it is possible to build a CDN-like subscription-oriented scalability service that provides on-demand scalability for dynamic Web content. Much like a CDN for static content, this scalability service will off-load requests for dynamic Web content from the content provider, correctly responding to new requests as the content provider's source data is updated.

This chapter continues by illustrating the potential benefits of a scalability service for dynamic Web content with several example scenarios in Section 1.1. Section 1.2 describes the traditional architecture for generating dynamic Web content, and Section 1.3 discusses the challenges in building such a third-party service to scale the traditional system design. Section 1.4 presents a high-level overview of our approach in creating such a scalability service, and Section 1.5 highlights the most important contributions of this thesis. Section 1.6 describes the organization of the remainder of this document.

1.1 Scenarios for a dynamic Web content scalability service

We illustrate the potential benefits of subscription-oriented scalability services for dynamic Web applications with three example scenarios: a conference management system, a budding electronic commerce Web site, and a civil emergency response system.

1.1.1 A scalable conference management system

Imagine the workload for an academic conference management system. After the conference announcement the system experiences only slight usage for many months, as the conference program committee forms and committee members and authors sporadically register for the conference. As the conference submission deadline approaches the system load increases as authors submit their work for review and publication: as authors rush to submit before the deadline the peak usage might be hundreds or thousands of times the typical system workload. After the deadline the system is again used only sporadically, with a second but more muted load peak as program committee members review the earlier submissions.

In this scenario, provisioning the conference management system for the peak load is extremely wasteful because the system load will almost always be significantly less than this peak. Currently the standard approach is to outsource conference management to a third-party provider, a service dedicated to academic conferences such as Microsoft's Conference Management Toolkit [68]. This solution, however, sacrifices control and customizability of the management system: the approach works well for conferences with a traditional cycle of paper submission, review, and publication, but constrains the interaction between authors and reviewers to match the system provider's pre-defined roles. A better approach would be simply to use a more general scalability service. Conferences that desire a traditional management interface could use a standard implementation of the management system, while conferences that require more flexibility could still implement their own management system cost-effectively.

1.1.2 An electronic commerce Web site

Consider a relatively small-scale Web-based e-commerce operation whose customer base is expanding. New customers bring more revenue, but may also lead to major management difficulties behind the scenes. Suppose the relatively low-cost equipment on which the e-commerce site was originally built (with the company's meager start-up funds) is becoming saturated with load, and will soon be unable to service all the customers. Standard solutions include upgrading to faster equipment on which to run the web, application, and/or database servers, or moving to a parallel cluster-based architecture as used by

big e-commerce vendors. Unfortunately, these solutions require a large investment in equipment, and, perhaps more significantly, funding for staff with the expertise necessary to manage the more complex infrastructure. Moreover, transitioning to a new architecture will undoubtedly create new bugs and may lead to costly application errors or system downtime.

A much more palatable option would be to subscribe to a scalability service on a pay-by-usage basis. A cost curve proportional to usage could potentially save the company large sums of money, especially if demand flattens out or drops. In this scenario, the equipment and management costs are shifted to the scalability service provider, where they can be amortized across many subscribers.

1.1.3 A new approach to civic emergency management

Finally, suppose the local government of a large city, such as Chicago, is ordered to prepare a response plan in case of a natural disaster. The government would like to have the capability of providing each citizen, even after the event has occurred, with both general and individualized instructions on how to protect themselves. In particular, the city would like to be able to provide maps and directions for each citizen explaining where to find medical treatment, shelter, uncontaminated food and water, etc. In addition, the city would like to be able to collect requests for immediate medical treatment from citizens who are immobile, and to collect reports from citizens and professionals about the effects of the incident in various sections of the city.

This application lends itself naturally to a Web-based implementation, but there are several inherent difficulties. First, demand for the application is likely never to occur, but if it should occur, it will come very quickly and in a very large dose. It would be costly for the city to invest in enough permanent infrastructure to satisfy the demand, and not cost-effective to keep this infrastructure idle. Second, it is critical to give all end users prompt and reliable access to information (recall the frustration of end users who were unable to retrieve even general news from http://www.cnn.com on September 11, 2001, because CNN was not utilizing the services of a CDN that morning [76]). It is even more important that data collected from end users requiring immediate assistance be recorded reliably. Third, the delivery of information customized to each end user, such as the generation of maps and directions, requires

significant computational resources. This information cannot easily be conveyed through a telephone conversation, and in any case, the scale of the demand would make a call-center solution impractical.

The ability to tap into a secure scalability service would alleviate these difficulties. The city would have to prepare software in advance and maintain a modest amount of permanent infrastructure, but could rely on the scalability service when demand suddenly arrived. The scalability service would then shoulder the network and computational load, even while potentially serving other, also critical, applications.

1.1.4 Common properties of the example scenarios

The example scenarios above illustrate the potential benefits of a plug-in scalability service for current and future dynamic Web applications on the Internet. These example applications share several key characteristics. First and foremost, the demand for each application is either unpredictable or varies over time. This property makes provisioning for the peak demand expensive compared to provisioning for typical system load, so building a dedicated system to handle the peak demand is not cost-effective. Second, each application generates Web content from both information in the user's request and information from some data repository, and as the data repository is updated, responses to new requests must accurately reflect the updates.

1.2 The traditional architecture for generating dynamic Web content

Figure 1.1 shows the traditional architecture for generating dynamic Web content. A typical Web application is a collection of programs, usually written in a procedural language like Java [44] or PHP [93]. Upon receiving a dynamic Hypertext Transfer Protocol (HTTP) [40] request, a Web server parses the request and forwards it to an application (app) server, which runs one or more of the Web application programs to generate the response. These applications usually interact with one or more data reposito-

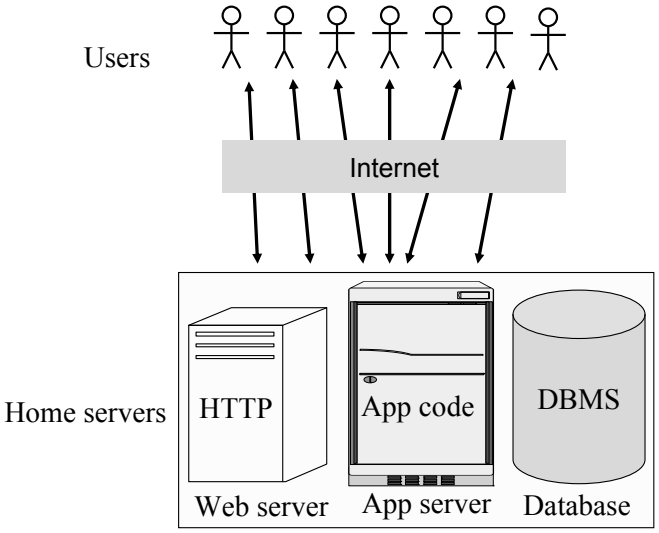


Figure 1.1: Traditional three-tiered centralized architecture for generating dynamic Web content

ries, typically by sending queries and updates to a relational database management system (relational DBMS or RDBMS) [32].

Overall, this design is called a *three-tiered architecture* and lends itself to a natural division of work: the Web server manages all HTTP interactions, the application server executes customizable application code and manages programmatic resource allocation, and the database server provides data persistence, rudimentary data processing, and reliability. In a traditional system, all three tiers are maintained at a central location by the content provider. We call this implementation the application's *home server*.

1.3 The challenges of scaling the traditional architecture

The key to scalability is to ensure that the home server remains lightly loaded even during periods of high request rates. In a traditional three-tiered architecture, the Web and application servers do not maintain any persistent mutable state and thus can be easily replicated so that each replica remains lightly loaded even at high request rates. At least one third-party CDN, Akamai, already provides this functionality [3].

The main challenge in scaling the traditional architecture is to scale the database component by off-

loading database work from the application's home server. Overcoming this challenge is made hard by several common difficulties:

1. **Content providers are reluctant to cede control of their data.** One appealing feature of the three-tier architecture is that it restricts all aspects of data management to a single component, the DBMS. This benefit is lost if the data is instead distributed and updated outside of the home server. This reluctance arises with good reason due to the security concerns, data corruption risks, and organizational difficulties entailed by distributed data management.

2. **New requests must reflect updates to the database as they occur.** To successfully off-load database work from the home server, the scalability service must respond to some database requests while requiring less work from the central database. A natural approach is for the scalability service to cache or replicate data on its own servers and respond to some requests without involving the home server. A key scalability challenge then is to propagate updates to the scalability service without creating additional work for the home server, while still allowing the scalability service to respond to a substantial fraction of the overall workload.

3. **End-user latency must not be substantially increased.** Most Web applications are interactive, and high user latencies are known to drive away customers [53, 54]. Moving work from the home server to a scalability service might increase the latency of some interactions between system components, since the scalability service and home server will often be located on different local networks. Imagine a Web application that executes many database requests for a single HTTP interaction. If the scalability service is located far from the home server, then these requests might increase the latency of the overall HTTP request, because they would otherwise have been local interactions had a DBSS not been used.

4. **Content providers must maintain control of data privacy.** In a world of well-publicized instances of data theft (e.g. [35, 55, 80]) and growing privacy laws (e.g. [25]), a content provider might be unwilling to risk revealing some data to a third-party scalability service. If possible, a scalability service must allow content providers to restrict access to some of their source data without unnecessarily restricting overall scalability.

5. **The scalability service must be architecturally compatible with traditional dynamic Web content generation.** The scalability service must execute Web applications designed for a traditional centralized three-tier architecture without requiring the content provider to redesign their system, learn any new programming languages or libraries, or otherwise reimplement their application. Any extra work required of the content provider is an undesirable barrier to their use of the scalability service.

Difficulty 1 precludes any solution in which the scalability service is the sole maintainer of some content provider data, and requires that the result of any updates eventually be sent to the content provider's home server. It does not necessarily disallow a solution in which updates are delayed or somehow aggregated, although such an approach might prevent the home server itself from responding to requests with up-to-date data. Difficulty 2 precludes caching techniques based entirely on timed data expiration, i.e. *time-to-live* (TTL) based protocols [33]. Unfortunately, full transactional consistency is well-known to present significant scalability challenges [46], suggesting that a successful scalability service will explore a middle-ground between weak TTL-based replication and full consistency techniques. With such an approach, a content provider must be able to restrict the ability of the scalability service to update critical data in a way that might violate consistency constraints.

For some applications, Difficulty 3 might prevent a fully distributed CDN-like approach and require the scalability service to use only servers near to each other or to the home server. Difficulty 4 is in some ways like Difficulty 2: whereas the latter might prevent the scalability service from itself executing some database updates, the former disallows the scalability service from itself observing the results of some database queries.

1.4 Our approach

We envision providing database scalability as a third-party *Database Scalability Service* (DBSS) provider, much as scalability for static content is currently provided by a CDN. A high-level illustration showing one use of our architecture appears in Figure 1.2. In our architecture users connect to proxy servers much

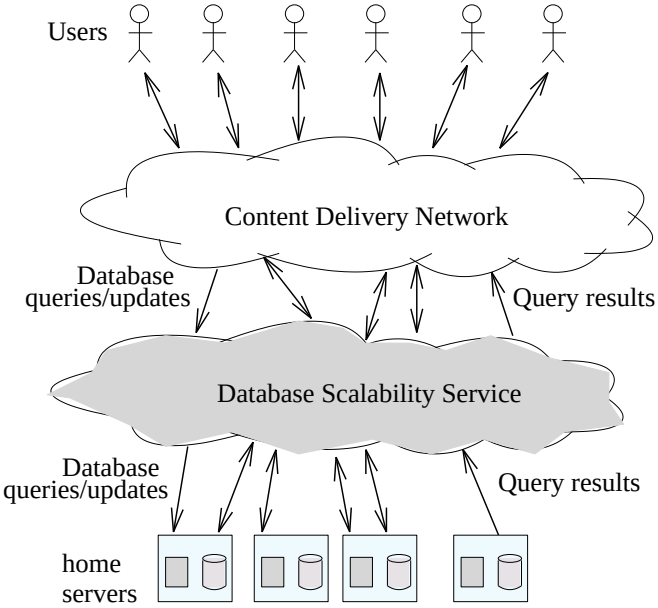
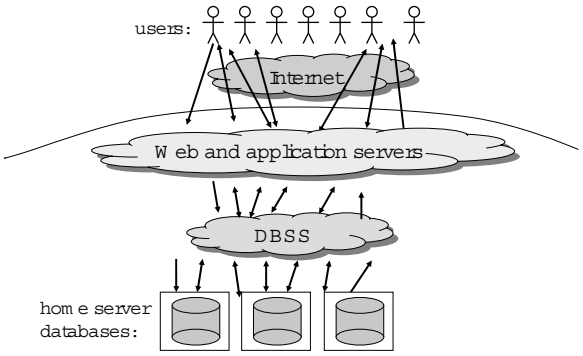


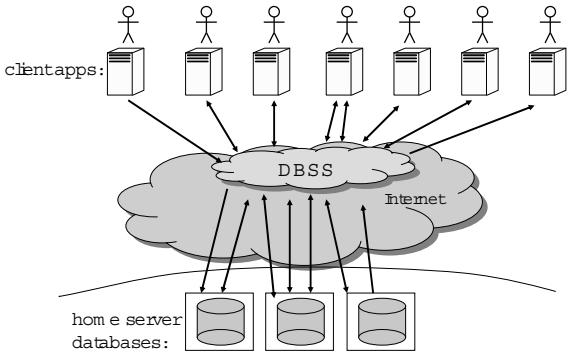
Figure 1.2: High-level illustration of one architecture using a DBSS

as they do for static content using a CDN. These proxy servers respond to Web requests and execute application code, like the Web and application servers of the traditional three-tiered architecture. Rather than connecting directly to a home database server, however, the application servers forward database requests to the DBSS, which instead encapsulates the data management tasks for the application. As with a CDN, the DBSS is a plug-in, pay-per-use service shared by many content providers, with a large shared infrastructure to absorb load spikes economically at any individual content provider.

One of our goals in designing such a service is that a DBSS could be used as part of any existing architecture that uses a database today, and that modifying an architecture to use the DBSS should require only minimal work. Figure 1.2 gave an example architecture in which a DBSS could be used. There, the Web and application servers are hosted by a CDN. Database requests are sent from the application server to the DBSS, which in turn interacts with the home server if necessary. This architecture is just one example architecture in which a DBSS can be used. Figure 1.3 illustrates several other architectures that use a DBSS. In Figure 1.3a, the DBSS is co-located on the same network as the home server, which also runs its own Web and application servers. In this scenario the content provider could potentially



(a) DBSS co-located with the content provider



(b) DBSS contacted directly by end users

Figure 1.3: Additional architectures that use a DBSS.

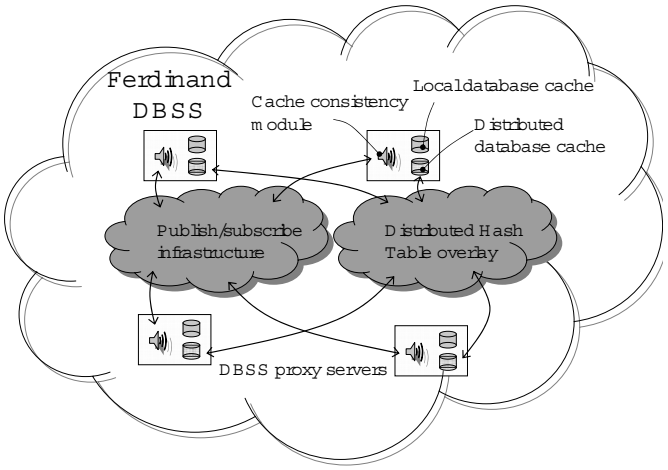


Figure 1.4: High-level architecture of the Ferdinand DBSS

use DBSS software on their own hardware, or perhaps the DBSS is still run by a third-party service at a shared network-hosting environment. Figure 1.3b shows an architecture in which the application code is executed locally by each end-user, perhaps within the Web browser on the end-user's computer. In this environment, the DBSS might exist outside the organization of the home server, either in its own shared, centralized network environment or distributed among many servers in a wide-area network. In all of these architectures, existing application code can be modified to use the DBSS with only minimal effort.

We have designed a DBSS called Ferdinand which is illustrated in Figure 1.4. Scalability is provided by caching data on behalf of the home database server, using both a local query result cache at each

DBSS node and a distributed cache implemented using a Distributed Hash Table (DHT). Cached data within Ferdinand is read only, and all database updates are forwarded to the home server for central processing. Ferdinand itself maintains consistency of the caches, which are notified of database updates via a fully distributed publish / subscribe system. The goal of the Ferdinand DBSS is to shield the home database server from queries executed by the application code, without placing additional administrative load on the home server.

This design allows us to address the difficulties listed in Section 1.3. Because the home database server contains a master copy of the database and all updates are forwarded to it, the content provider maintains control over their data. Update notifications are immediately published and delivered using the publish / subscribe system, so new requests reflect updates as they occur. Our use of a simple query result cache rather than a more complex representation requires minimal processing within Ferdinand, enabling strategies to reduce end-user latency and control data privacy with encryption in the cache. Finally, the Ferdinand DBSS can be implemented as a middleware service providing the same interface as a typical database. To use Ferdinand a content provider typically needs to change just one line of their application code, to use the Ferdinand database driver rather than the database driver of the home database server.

In this thesis we focus on scalability-related aspects of the Ferdinand DBSS. Security- and latency-related issues of Database Scalability Services are examined in a recent thesis [65] by Amit Manjhi.

To provide database scalability the Ferdinand DBSS exploits several key properties typical of Web applications: (1) the underlying data workloads tend to be dominated by reads, not writes, to the database, (2) queries and updates are often simple and do not involve complex data relationships, and (3) the range of database requests executed by a given Web application is often limited by a small, fixed set of query and update templates in the application code. The first property makes it feasible to handle all updates at each application's home server. The second and third properties enable us to exploit foreknowledge of the Web application and its data to improve scalability for the range of requests an application actually executes.

The Ferdinand architecture contains two core components that closely relate to its scalability: the

query result cache, and the publish / subscribe infrastructure for consistency management. This section continues by discussing each of them in turn, and then describes our work to better exploit foreknowledge of the application code, a key technique we use to improve scalability.

1.4.1 Scalable query result caching for Web applications

The first key innovation of our Ferdinand design is our database query result cache. As mentioned above, we chose this design for its simplicity. Implemented as a map between database queries and their query results, our cache requires no parsing of SQL queries or results, uses well-understood data structures, and can be used even when queries and results are encrypted to prevent the scalability service from observing application data.

This simplicity has disadvantages, however. The biggest disadvantage is that without any query processing, our cache only produces results for exact query matches. Our cache cannot respond to highly similar queries that would produce the same result, or combine results to respond to new queries that could be answered with data already contained within the cache. Our cache design also uses storage inefficiently. Because overlapping query results might duplicate base data, the size of the query cache could potentially exceed the size of the home database. These disadvantages exacerbate the key challenge of the scalability service, to maintain a high cache hit rate while enforcing strong consistency management.

Our solution is to use a two-tiered cooperative cache in which a Ferdinand DBSS server checks both a local disk-based cache and a master cache at a remote node before forwarding a request to the home server. This design has the potential to greatly reduce the load at the home database, but it introduces new complications. First, the latency for a cache miss can be higher than with a standalone cache since the request is forwarded to a remote Ferdinand node before to the home server. Second, our tiered cache design further complicates consistency management.

In this thesis we design and implement such a cooperative cache and study these design trade-offs. We design and implement a consistency management system that overcomes the additional complexity of a two-tier cache, and examine the two-tiered cache in various network environments. Overall, we

show that database query result caching is a feasible approach that successfully scales data access for Web applications.

1.4.2 Publish / subscribe for consistency management

The second key innovation in Ferdinand is our use of publish / subscribe for efficient consistency management. Our goal is to ensure that each cache receives notification of any database update that affects a cached query result, while not relying upon any central infrastructure at the home server and reducing communication among the caches.

The update notification problem is a natural application for publish / subscribe. When a cache stores a query result it generates subscriptions related to that query result. Each update to the database is published, and the publish / subscribe system matches the update to the query results it affects. The publish / subscribe system therefore encapsulates the process of matching updates to affected queries and notifying affected caches of the updates.

The key advantage of using publish / subscribe in this setting is that it allows us to leverage existing technology for the communication of updates to the Ferdinand DBSS caches. However, it introduces several implementation challenges. First, there are a wide variety of publish / subscribe systems, supporting different degrees of scalability and different methods of matching publications to subscriptions. Matching database updates to affected queries is more complex than the matching process supported by most existing publish / subscribe systems, and it is not clear what type of publish / subscribe system we should use. Second, once we have selected a particular publish / subscribe system, it is not clear what publications and subscriptions should be made to ensure that a DBSS node is notified of all updates that possibly affect its cached queries.

In this thesis we show that even the simplest publish / subscribe paradigm – topic-based publish / subscribe – is sufficient to implement consistency management for Web application database query result caches. The key idea of our approach is to use foreknowledge of the Web application and its database requests to constrain the range of queries and updates that the Web application might execute. Constraining the range in advance allows us to efficiently support update notification for those queries

and updates, at the expense of potentially inefficient operation for queries and updates that the application never executes.

We design and implement such a system using topic-based publish / subscribe, demonstrate that it is sufficiently scalable for a DBSS's consistency management needs, and show how to use publish / subscribe to implement consistency management for our two-tiered cooperative caching system.

1.4.3 Exploiting foreknowledge of the Web application

To efficiently implement consistency management using publish / subscribe we use exploit foreknowledge of the Web application, using an offline analysis of the application and its database requests. Because the database requests in many Web applications are written as a small number of database templates with parameters that are instantiated at run time, we can analyze the relationship between the application's database queries and updates to determine which updates affect each query.

A more important relationship, however, is the relationship between the application's updates and the application itself – not the relationship between the application's updates and queries. Our above analysis explored just the relationship between the query and update templates themselves, and did not consider the broader use of their data by the application. For many simple query and update templates, the relationship between application data and the source DBMS data is defined at compile time, not run time. We use compiler-like data propagation across the application-DBMS boundary to extend our offline analysis to what we call a *holistic analysis* of the Web application – an analysis of the Web application and its associated database requests that transcends the traditional programmatic boundary between the application and DBMS – while still maintaining that traditional boundary for the programmer.

Our holistic analysis allows us to identify situations in which an application's update template potentially affects a query template, but the update can not affect the application's execution. We identify two specific cases where this is true. First, the update might affect only data not used by the application, even though that data is retrieved by a query the application executes. In the second case, some instantiations of an update template might affect the application, but those instantiations are never executed by the application.

We use holistic analysis to refine the relationship between the Web application and its updates, and apply this analysis to the publish / subscribe-based consistency management infrastructure of the Ferdinand DBSS. We compare our holistic analysis to the offline analysis that just examines the templated requests, and show that holistic analysis reduces the traffic needed for correct consistency management within the scalability service.

1.5 Contributions of this thesis

The most important contributions of this thesis are the following:

- We design and build the first cache-based third-party scalability service for dynamic Web content, featuring tunable privacy and fully decentralized cache consistency management. **(Chapter 3)**
- We design and build the first multi-level cooperative cache for database query results, a key part of the Ferdinand scalability service. **(Chapter 3)**
- We show that cooperative query result caching can scale the generation of dynamic Web content by more than a factor of 10 over centralized solutions and as much as a factor of 3 over standalone caches, demonstrating that query result caching is a feasible technology for scaling the database. **(Chapter 5)**
- We provide the first quantitative demonstration that broadcast-based consistency management can be the primary performance limitation in a database caching system, even when the overall system performance is otherwise limited by forwarding all updates to a central database server. **(Chapter 6)**
- We show how to efficiently communicate database update notifications to query result caches using topic-based publish / subscribe and explore several alternative ways of doing so. **(Chapter 6)**
- We introduce offline techniques to examine the dependences between a database application and

its database requests and show how this analysis can be used to improve the efficiency of communicating update notifications within a DBSS. **(Chapter 7)**

1.6 Organization of this thesis

We continue in Chapter 2 by discussing the previous work related to our approach. Chapter 3 describes the Ferdinand architecture and implementation, the first scalability service for dynamic Web applications. Chapter 4 describes our methodology for evaluating a DBSS, including a description of our benchmark applications and experimental environment.

In Chapter 5 we evaluate Ferdinand, comparing it to multiple alternative architectures, including a DBSS design that does not use cooperative distributed caching. We explore the performance implications of the two-level cache design in several network environments, evaluating which environments are best for Ferdinand compared to the alternative scalability service designs, and show that Ferdinand's overall performance can far exceed that of the alternative systems.

Chapter 6 details our use of publish / subscribe to maintain the consistency of Web database caches. We discuss the potential design space for using publish / subscribe in this setting and describe how we apply an offline analysis of the Web application's database requests to efficiently use publish / subscribe for cache consistency management. We introduce several alternative approaches for how to use that offline analysis, evaluate the trade-offs of each approach, and measure the extent that publish / subscribe-based consistency management reduces the number of consistency messages and increases system scalability compared to the more primitive method of broadcasting updates to the scalability service nodes. We build a simulator of networked caches to examine the performance of publish / subscribe for consistency management in a wide variety of network environments.

Chapter 7 introduces application-update dependence analysis, the study of whether a database update affects the execution of a database application. Application-update dependence generalizes our offline analysis from Chapter 6. We apply application-update dependence analysis to our Web application benchmarks and compare its performance to traditional query-update dependence analysis both statically

and when used to configure Ferdinand's consistency management infrastructure.

Finally, Chapter 8 summarizes the main results of this thesis, discusses their implications, and highlights important problems that we leave open as future work.

Chapter 2

Related work

Our work touches on many sub-areas within computer science including databases, file systems, networks and publish / subscribe, and middleware. In this section we describe the work most relevant to ours. We start by briefly presenting the historical context of database transactions and consistency management, and then discuss other work on improving database performance and scalability. We then discuss other approaches to scaling Web content delivery, with a focus on the generation of dynamic Web content. We also discuss relevant work on publish / subscribe systems, which we use to propagate update notifications to database query result caches. Finally, we discuss work related to our offline analysis of database applications which we use to optimize our communication of updates by the publish / subscribe system.

2.1 Database transactions and consistency management

The modern theory of database systems was largely defined in the late 1960's and early 1970's, including the notion of database consistency and the serializability, isolation and durability of concurrent transactions [38]. The theoretical foundations of consistency were later expanded by Papadimitriou, Bernstein, and others for replicated database systems, developing the notion of one-copy serializability [19, 75]. Meanwhile, there was a flurry of work on consistency management for distributed database systems, in-

cluding distributed consistency management algorithms based on two-phase commit [6, 92], timestamp ordering [87], majority consensus [94], and other methods. Bernstein et al. provide a good overview of these topics in [18].

The conflicts between consistency, availability, and performance in the face of failures have long been known to database researchers [17, 45, 77]. To complicate matters, as the size of the system scales and the number of concurrent transactions increases, the chance of conflict between transactions often increases disproportionately [46, 47].

The trick of sacrificing consistency to gain performance is a well-established strategy in the database community [41], and with the advent of large distributed systems, various notions of relaxed consistency have gotten greater attention recently. Gray himself advocated a strategy of allowing so-called *tentative transactions* to improve performance and availability [46], and Bernstein et al. recently examined a relaxed consistency model in which each read operation specifies a time-based freshness constraint, allowing transactions to use data that is somewhat out-of-date [16]. In [99] Yu and Vahdat explore a richer model of relaxed consistency that allows flexible bounds not only in how out-of-date data can be, but also enables bounds on how inaccurate in value data can be, or how many times the data has been written since its claimed value was up-to-date.

Our work adopts this approach of sacrificing some consistency to achieve greater scalability. We focus on minimizing how out-of-date a query result can be, attempting to reduce the chance that a cached result is stale when it is used. We additionally guarantee that our scalability service provides a fully consistent, one-copy serializable execution if only single-statement transactions are used. Finally, we allow the content provider to specify full consistency for arbitrary transactions, enabling them to implement serializability for critical transactions that must be kept fully consistent and not merely up-to-date. As you will see in Section 2.2, the level of consistency provided by our scalability service is similar to that of many other approaches to scaling the generation and delivery of dynamic Web content.

2.2 Database scalability for dynamic Web content

Scaling the back-end database specifically for dynamic Web content is a well-studied problem, studied both by commercial efforts and academic research projects. The common approaches can be broken into three basic categories: database replication, database caching, and database outsourcing. The remainder of this section discusses each of these in turn.

2.2.1 Database replication

Database replication is the strategy of scaling the database by explicitly copying all or part of the database to more than one server. Most replication implementations use a *read-one-write-all* policy, in which a read can occur at any copy of the data and database updates are forwarded to all data copies. A read-one-write-all policy combined with full replication easily scales database reads, but updates become a significant bottleneck since every update must be broadcast to every server. Partially replicating the database can better scale database updates since each update must be broadcast to only some servers. A challenging additional problem for partially replicated systems is how to specify the replica *data layout*, i.e. where, and at how many nodes to copy each piece of data to handle database queries and updates efficiently.

GlobeDB [88] uses partial replication without full distributed consistency management. Like the back-end database server in our system, GlobeDB also has a *master* server that contains all data in the database and receives all database updates. The presence of a master server also simplifies query processing: if no single replica contains all the data needed to respond to the query, it can simply be forwarded to the master server. To avoid the need for a full distributed consistency management implementation to achieve transactional consistency, they relax consistency to achieve the same consistency model that we do: full one-copy serializability for single-statement transactions but reduced consistency for multi-statement transactions. GlobeDB's main contribution is in how the replica data layout is determined. Rather than require a human expert to manually specify what data should be copied to each replica, GlobeDB automatically determines the data layout by inspecting a trace of database accesses,

finding clusters in the data used by those accesses, and replicating the clusters that are frequently used.

GlobeTP [48] is derived from GlobeDB and uses a similar architecture based on partial replication with a master server, and provides the same consistency guarantees. Its main advance over GlobeDP is an improved algorithm to automatically partition the database. Instead of examining a trace of the database requests executed by the application, GlobeTP uses an offline analysis of the dynamic Web application to predict the queries executed at runtime, allowing them to determine clusters of data accesses and replicate the data as with GlobeDB. Their offline analysis is highly similar to the offline analysis we use to efficiently communicate update notifications.

Microsoft has also implemented a replication-based product for scaling dynamic Web content, MTCache [57], specifically for use with Microsoft SQL Server [67]. Unlike GlobeDB and GlobeTP, MTCache requires a database administrator to explicitly specify what tables (and parts thereof) are replicated at each MTCache server. MTCache uses a cost-based optimizer to decide whether to execute a query at a replica, at the back-end server, or partially at both. MTCache also allows the database programmer to specify an arbitrary freshness constraint on the source data used to answer a query, but like our scalability service, only guarantees one-copy serializability for single-statement transactions. To use MTCache's freshness constraints, a database programmer must modify the application to use MTCache's SQL extension. Unlike our scalability service, MTCache centrally broadcasts updates to each replica server.

2.2.2 Database caching

An additional problem with database replication is that the replica data layout is static: it does not easily adapt to changes in the database workload over time. Database caching is an alternative approach without this problem. With database caching, the results of database queries are stored and can be reused at remote nodes without involving the back-end database server. Compared to explicit database replication, database caching is highly dynamic. Caches automatically adapt to the database workload, without requiring the intervention of a database administrator. Database caches, however, introduce the problem of efficiently keeping the caches up-to-date as the database is updated. The simple solution of broadcasting each update to each cache can limit overall scalability.

A large number of projects have used some form of database caching as a means to scaling the delivery of dynamic Web content. IBM's DBCache [62] uses an architecture similar to ours, in which proxy servers generate dynamic content and locally cache database requests. DBCache's representation of query results within the cache is much more sophisticated than ours. They introduce the notion of *cache tables*, using IBM's general database, DB2 [52], to store the base data from which results are derived, much like MTCache. This enables them to store more results with less storage than our system, but requires each proxy server to implement the complex query processing of a standard database and is also dependent on the implementation of the back-end database server, which must run DB2. Like our initial prototype implementation, a DBCache proxy server can only respond to queries that it has exactly seen before, limiting the overall cache hit rate. Also, DBCache relies on an inefficient centralized broadcast mechanism to propagate every update to every cache. Like GlobeDB and GlobeTP above, DBCache implements the same consistency model as ours, with one-copy serializability only for single-statement transactions.

IBM also studied database caching for dynamic Web content in another project, IBM DBProxy [11]. DBProxy's main contribution is their sophisticated query processing methods. Like DBCache, DBProxy stores cached query results in cache tables. DBProxy, however, implements general query containment algorithms to decide if a query can be answered from cached source data, even if the exact query has not been answered before. If a query can not be answered locally because some source data is missing from the cache, DBProxy transforms the query to retrieve only the missing data. Overall, these query processing strategies can improve the cache hit rate and further reduce the load on the back-end database server. DBProxy implements the same consistency model as us but, like DBCache, centrally broadcasts updates to keep the caches up-to-date. In [12], Amiri et al. extend DBProxy's invalidation system, creating a publish / subscribe-like system much like our own. To do this, they use a central online analysis of the database workload to extract template information from the database request stream, creating a system of centralized filters to prevent sending unnecessary update notifications to the proxy caches. This solution has the advantage of reducing the load of unnecessary invalidation messages from the proxy server, but since the filtering and processing of invalidations occurs locally at the central database, its centralized consistency management remains a potential bottleneck for overall system performance.

NEC implements a product, CachePortal [59, 60], very similar to IBM's DBCache. It enables dynamic Web content generation at proxy servers at the network edge, using a database cache to respond to database queries if possible and otherwise forwarding the request to the central server. Like DBCache they also use centralized broadcast-based invalidation, inspecting the database log and forwarding updates to each database cache. Their main contribution is in providing adaptive freshness constraints: as their system becomes overloaded, it intentionally delays sending updates to the caches, allowing greater scalability but at the cost of staler data.

MySQL [69] implements a simple cache service to be used with their back-end product. Like our scalability service, MySQL's cache is a map of database queries to their materialized results, allowing the back-end server to respond to queries without using the full database engine if the query result is present in the cache. Their cache, however, suffers from two severe limitations. First, the cache is only available at the central database server, and thus does not provide the potential scalability of a distributed caching product. Second, to ensure consistency they use a table-based invalidator of cached entries: if an update affects any part of a database table, they invalidate all query results that depend on that table even if the query result is not affected by the update. This allows them to easily ensure that all affected query results are invalidated and provide full one-copy serializability for all transactions, but provides only limited scalability and a low cache hit rate.

Amza et al. have extensively studied database replication and caching for dynamic Web applications [13, 14, 63, 64, 89, 90, 91]. Of these the most closely related to our work is [14], in which they study many basic issues in a cache-based system such as ours. They show that fine-grained invalidation is essential for good cache performance, but that the placement of the cache at the web server, database server, or a standalone cache server is largely irrelevant when resources are otherwise unconstrained and cache hit rate is the same. They also explore a system with a two-level cache, in which a local cache is first inspected and cache misses are sent to a central caching server. Overall, however, their system results in cache hit rates far below that attained by our two-level cooperative caching system, and they rely on centralized broadcast-based invalidation for consistency management.

2.2.3 Database outsourcing

A third approach for scaling the database for dynamic Web content is to entirely outsource all aspects of the database management to a third party. This approach suffers from Difficulty 4 as discussed in Section 1.3: content providers are often unwilling to risk compromising the privacy of their data. Because of this, much of the research in this area has focused on privacy rather than on scalability.

In [51], Hacigümüs et al. describe an outsourcing approach in which the outsourced database is encrypted, preventing the third-party provider from accessing the database arbitrarily. The challenge with this approach is to efficiently provide the query processing capabilities needed by typical Web applications. In [49] and [50] they provide techniques that enable SQL queries on encrypted data, examining the performance cost of encrypting data within the database and showing how privacy homomorphisms [82] can enable the efficient computation of some SQL aggregate queries on encrypted data. Currently, however, techniques such as privacy homomorphisms lack a rigorous security analysis and have not yet been shown to be effective in practice for database outsourcing [39]. Overall, Hacigümüs et al.'s approach focuses on privacy rather than scalability.

Amazon.com, Inc. [7] has recently introduced two data-related web services, Amazon Simple Storage Service (Amazon S3) [9] and Amazon SimpleDB [10]. Both of these services completely eschew privacy and focus on scalability and availability. However, they both offer a less sophisticated data model than that of the relational DBMS used by a typical dynamic Web application. Amazon S3 provides a simple object storage mechanism, essentially providing a map between an object's name and its data. Amazon SimpleDB is more like a typical RDBMS, providing full relations and allowing simple queries to be executed on those relations. However, SimpleDB does not implement full SQL queries, and in particular does not allow join queries or any queries that access multiple relations, functionality commonly used by dynamic Web applications.

2.3 Non-database-centric approaches to scale dynamic Web content

A number of projects have sought to scale dynamic Web content generation using approaches that do not focus on the database. Of these, the most successful approaches restructure the Web application or avoid execution of the Web application altogether – and its database requests – by caching the output of the Web application. These approaches, however, do somewhat violate Difficulty 5: since a Web application is not necessarily executed to respond to each Web request, these strategies diverge from the traditional three-tier architecture and can cause behavior not expected by the Web application programmer. For example, if a Web application's output was cached and reused to avoid re-executing the application, any side effects of that application would not correctly occur.

Of these, the most relevant and prolific work has been done by IBM research scientists Challenger, Iyengar, and Dantzig. In [29], Challenger et al. describe a dynamic Web content cache that uses knowledge of the dependence between each dynamic Web page and the source data, purging affected pages from the cache when the relevant source data is updated. They cached the entire output of each dynamic Web page and updated modified pages within the cache rather than invalidating them, achieving very high cache hit rates and avoiding execution of the Web application. They later extend their work in [28] to a similar system based on caching fragments of dynamic Web pages rather than whole pages. Like our work, they utilize a publish / subscribe-like system to notify each cache of updates to a dynamic Web page or page fragment, using the known dependences between the source data and each cached object. Unlike our work, their publish / subscribe architecture is centrally based, and their caches do not cooperate with each other to improve overall cache hit rates. Also, they require the application programmer to provide the object dependence graph relating each Web page to its source data.

In [26], Chabbouh and Makpangou extend Challenger et al.'s work to automatically fragment dynamic Web pages, using an automated static analysis to determine the relationship between the page fragments and the parameter values given in the address of thy dynamic Web page. Using their fragmentation technique, they extended their work to create a fragment-based CDN-like scalability service in [27].

In [56], Labrinidis and Roussopoulos describe techniques to maximize a Web application's data freshness in the face of high update loads. Generally, a system for generating dynamic Web content faces one of three options when an update occurs: (1) merely update the source data within the database, (2) propagate the update to immediately generate the materialized views used by the Web application, or (3) immediately regenerate the dynamic Web page by executing the Web application. They provide an analytical framework to determine which strategy should be used for given a Web application and update load. Their work is highly relevant but orthogonal to ours. In our scalability service we always use approach (1), updating just the source data but invalidating any cached materialized views affected by the update. Although we did not do so, a similar analytical framework could be applied to our scalability service to determine whether a cached materialized view should be updated or merely invalidated, or perhaps even whether the dynamic Web page should be immediately regenerated.

Akamai [2], a major content delivery network for static content, implements an EdgeJava [3] product that scales the application server to generate dynamic Web content at the edge of the network. EdgeJava provides a database cache to offload some work from the central database server, but provides only weak consistency for cached data based on a combination of timed data expiration (TTL) and "do-not-cache" directives. While EdgeJava has been used by some applications that do not rely heavily on the database (e.g. [4]) we are not aware of any uses of this service for traditional data-intensive Web applications.

More recently, Amazon's Elastic Compute Cloud (Amazon EC2) [8] provides on-demand scalability for general computing, which could conceivably be used to implement on-demand scalability for dynamic Web applications. While general computing platforms are ideal for scaling the Web and application servers – for which computation is the performance bottleneck – it is not clear if such an environment can successfully scale the database component, for which data access and limits on concurrency are typically the performance bottlenecks. One feasible approach might be to implement a scalability service like ours in their flexible environment, much as we implemented our system experiments on Emulab [98].

Finally, a number of projects seek to scale dynamic Web content generation by fundamentally changing the architecture used to generate it. In [43], Gao et al. describe an edge service architecture based

on distributed objects, each of which provide different levels of concurrency and consistency semantics. In their system Web content is generated at the network edge as it can be with our scalability service, but rather than access a centralized database, the core application data is stored by distributed objects throughout the network. In their description these objects are application-specific and designed for the TPC-W benchmark, although their object specifications can generalize some types of data interactions common to many Web applications. A recent project by Wei et al. [97] similarly decomposed the TPC-W and RUBiS benchmarks into limited data-oriented Web services [22], and then showed that traditional scalability approaches applied to those individual services could scale the overall performance to greater throughputs than the same methods applied to the benchmarks as a whole. Overall, we find both of these approaches to be very promising, especially as Web service architectures gain increasing popularity as an industry practice. Unfortunately, approaches such as these require significant extra programmer work when applied to existing applications: to implement an existing application as a Web service architecture, one would typically need to entirely redesign and reimplement the data-related logic of the application.

2.4 Publish / subscribe for consistency management

One key aspect of our approach is our use of publish / subscribe to notify the proxy servers of database updates. In doing so, our goal is to ensure that each cache receives notifications for all updates that might affect it, while simultaneously minimizing the number of unnecessary notifications that each cache receives. Several earlier projects have used publish / subscribe-like techniques for consistency management or similar database problems.

Group communication has long been among the tools used to facilitate database replication. Early approaches often required synchronous, broadcast communication, even though database researchers espoused that such methods were impractical for real systems [46]. Alonso extensively explored providing different degrees of transactional serializability with limited communication guarantees. Of his work, the most relevant to publish / subscribe is one of his early theoretical results, in which Alonso showed that order-preserving serializability[15] at all replicas and reliable in-order delivery of transac-

tion messages are sufficient to guarantee serializability for partially-replicated databases [5]. This result is highly similar to what we attain with our scalability service: our theoretical results assume weaker properties (order-preserving serializability at the central database server only, reliable but possibly out-of-order message delivery) and show that stale data will be correctly purged from all caches even though one-copy serializability is not guaranteed.

In [30], Chandramouli et al. describe a publish / subscribe system in which publications are matched to subscriptions based on state contained in an external database, and build a system in which publications and subscriptions are SQL queries with a notification occurring if a published update affects a query result. Although their goal is to ensure notification whenever a continuous query result changes, their work applies very directly to our problem of ensuring update notifications to query result caches. There are several key differences, however, between our applications. First, our ability to analyze Web applications offline allows us to constrain the range of queries and updates that will be used as subscriptions or publications during system execution. This analysis gives us the potential to match publications to subscriptions more efficiently than in their system. With our query result caches, however, we expect that each cache contains more entries than would typically exist as continuous queries in their system, meaning that our publish / subscribe system must handle a higher rate of subscriptions and unsubscriptions than their own.

As mentioned in Section 2.3, Challenger et al. use a publish / subscribe-like system for cache consistency management in [28] and [29]. Although they use their system for the same general purpose as ours – maintaining cache consistency for dynamic Web content – their system has a vastly different architecture than ours. First, their publication system is centralized: they use database triggers associated with the source data to initiate the publication process. When a trigger is activated, the newly-modified source data is published to a module which determines which Web pages (or fragments) are affected and reconstructs those pages. Second, in their architecture each cache contains copies of all or most generated Web content and receives every update notification. In other words, they do not use publish / subscribe like us to filter unnecessary update notifications from the caches, and actually broadcast all notifications to every cache. Instead, they just use their publish / subscribe-like system internally at each cache as a mechanism to map update notifications to the particular pages that are affected by the update.

2.5 Application-level analysis for database applications

Finally, one of our key techniques is to analyze the database application to determine dependences between the application and the database requests it executes. Our project draws upon two key areas: traditional query-update dependence analysis within the database community, and more directly from recent work that also examines similar static properties of database applications. We describe these areas in turn.

2.5.1 Database query-update dependence analysis

The problem of whether a database application is affected by an update is closely related to the problem of whether a database query is affected by an update, a well-studied problem in the database community. While this problem is undecidable in general, it can be solved for many common types of query-update pairs [20, 37, 58].

Blakeley et al. [20] and Elkan [37] gave efficient methods for determining query-update independence that relied upon showing that the query's result was not derived from data affected upon the update.

Levy and Sagiv later gave a more general method, reducing the problem of query-update independence to the problem of determining equivalence of datalog programs [58]. Their method constructs two datalog programs: one datalog program of the query executing on the original database, and a second program of the query executing on the modified database with the update applied to it. They proved that if these two programs are equivalent, then the query and update must be independent since the update does not affect the execution of the query. They also showed that if a query and update are data-independent (as in [20] and [37]) then the relevant datalog programs are easily proven equivalent, so that program-equivalence is strictly more powerful than data-independence for proving query-update independence.

The insights of both the above theoretical techniques – data-independence and program-equivalence – apply to the application-update independence problem. As with Blakeley and Elkan, if an application is independent of the data affected by an update, then the application and that update are necessarily

independent. More generally, like Levy and Sagiv, if the application's execution before the update can be proven equivalent to the execution after the update, then the application also must therefore be independent of the update.

In practice, the application-update dependence problem differs from the query-update dependence problem in two key respects. First, a database application often does not specify the exact queries and updates it executes but instead specifies query and update templates which define a range of database requests it might execute. The application-update independence problem therefore often reduces to the question of whether *some set* of possible database queries is independent of some set of possible updates. Second, even if an application uses a database query affected by some update, the application itself might not use the affected data and therefore might still be independent of the update. Our work does not significantly extend the theory of query-update dependence analysis, but we examine the implications of these differences further in Chapter 7.

2.5.2 Offline analysis of database applications

The second key aspect to our application-update dependence analysis is the analysis of the database application itself. The main idea of our approach is that we use automated analysis to better understand the data dependences between the application code (written in a procedural language) and the SQL database requests.

The problem of mapping application data to DBMS data has long been a data management problem. Early approaches such as object-oriented databases and persistent programming languages failed to gain wide acceptance in the database community, and common current practice is still to manually program the database access logic or to use an object-persistence layer (e.g., Hibernate [81] or TopLink [73]) to manage the mapping between application and database data. Recently, Melnik et al. [66] proposed a model in which the programmer explicitly declares a general-purpose mapping between application and database data, independent of the programming model and DBMS used. A significant drawback of their approach, however, is that it requires extra work for the application programmer, who must explicitly declare the data mapping. We favor solutions that do not require additional work for existing

applications.

Chabbouh and Makpangou implemented an offline analysis of PHP Web applications to detect dynamic page fragments generated by the application [26]. In doing so, they detect the dependence between the Web application's parameters and the output produced by that Web application. They also discuss using a similar analysis to relate the page fragments to the database source data from which those fragments are derived, but they do not claim to have implemented this functionality or discuss its implications. This latter analysis would be very similar to our offline analysis: we relate the source database data to the application data, which they could then relate to the dynamic page fragments.

A similar relationship between Web pages and their source data is used by Challenger et al. in [28] and [29], but they do not automatically detect these relationships and instead depend on the application programmer to provide them. In [31], Choi and Luo used yet another similar analysis to map between a dynamic Web page's URL and the SQL requests it generates, also for the purpose of maintaining dynamic Web page caches. Overall, none of this work considers the use of their intermediate data analysis for use by other optimizations.

Finally, in a related thesis, [65], we use offline analysis across the application-database boundary to optimize database applications for execution in high-latency distributed environments, such as the scalability service we build here. There we used the offline analysis to to recompile Web applications to require fewer database requests while generating equivalent output, reducing the average execution time of the applications in high-latency environments. Although both that work and this thesis examine data across the application-DBMS boundary, there we did not explicitly map DBMS data to application data, and our use of the offline analysis targets different subproblems for executing Web applications in distributed environments.

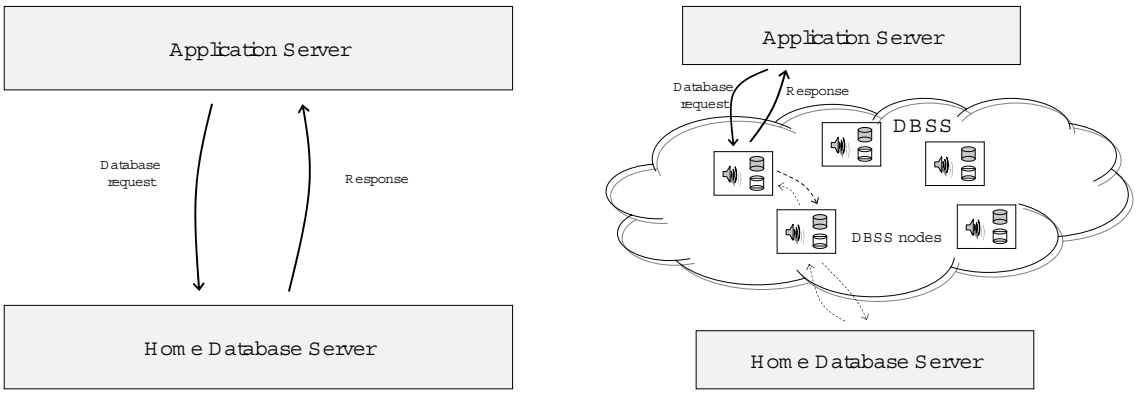
Chapter 3

Architecture of the Ferdinand DBSS

As described in Section 1.4, we envision a third-party Database Scalability Service (DBSS) provider that scales the database component of the traditional three-tiered architecture much like CDNs already scale the Web server for static content. The DBSS is designed to be fully compatible with the traditional three-tiered architecture, used much as if the DBSS were inserted between the application server and central database of a traditional system.

Figure 3.1 highlights the path of a database request in both a traditional architecture and in a system using a DBSS. In the traditional system (Figure 3.1a), the application server generates a database request and sends it directly to a home database server, which responds to the request. Using a DBSS (Figure 3.1b), the application server instead sends the database request to a DBSS node. The DBSS node responds to the request directly if possible, possibly contacting other DBSS nodes in the process. If the DBSS cannot respond to the request itself, it forwards it to the home database server.

In this thesis we have designed and built a DBSS implementation that we call Ferdinand, as introduced in Chapter 1. There we presented a high-level overview of its design, which we illustrate (Figure 3.2) and repeat here. Ferdinand provides database scalability through a collection of cooperating query result caches. Database queries are cached both locally at each Ferdinand node, and also in a distributed cache provided by a Distributed Hash Table (DHT). All updates are forwarded directly to the home database server. Ferdinand maintains the consistency of the query result caches without needing interaction from



(a) Database request path without a DBSS.

(b) Database request path with a DBSS.

Figure 3.1: Path of a database request with and without a DBSS. Without a DBSS all requests are sent directly to the home database server. With a DBSS the requests are instead sent to a DBSS node. The DBSS node might forward the request to other DBSS nodes, and only some requests are eventually forwarded to the home database server.

the home database server, using a publish / subscribe system to notify Ferdinand nodes of updates as they are received.

This chapter continues by describing the Ferdinand-related modifications necessary to the content provider's home database server and application server in Sections 3.1 and 3.2, respectively. In Section 3.3 we then describe the architecture of the Ferdinand DBSS node itself, our core contribution to the Ferdinand design. After describing each of Ferdinand's components, we describe the main algorithms we use to implement Ferdinand in Section 3.4.

3.1 The home server

To use our scalability service, the content provider's home server requires one minor modification: the home server must run a user-level application, the Ferdinand Home Server Module (HSM), shown in Figure 3.3. The HSM is a lightweight Java application that accepts connections from Ferdinand nodes and on their behalf executes database requests locally at the home server. The content provider must

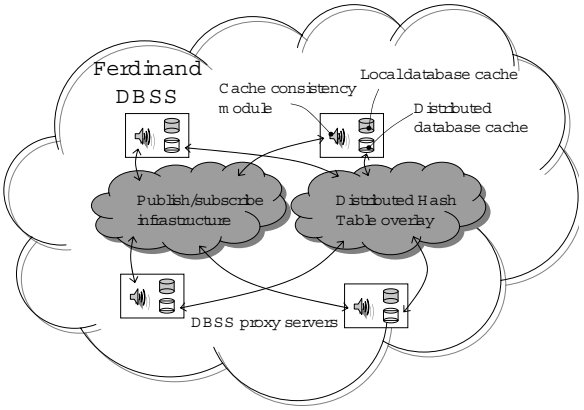


Figure 3.2: High-level architecture of the Ferdinand DBSS (repeated from Figure 1.4)

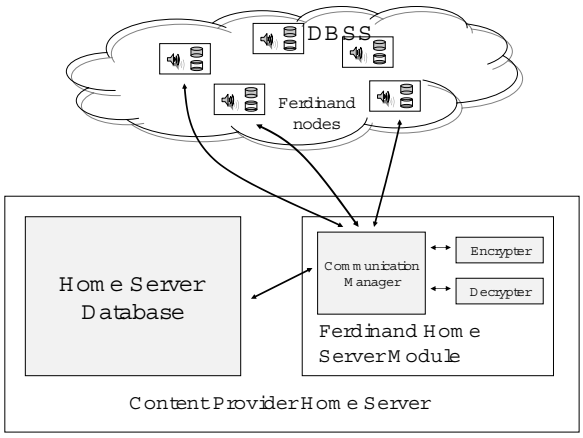


Figure 3.3: The architecture of the home server with the Ferdinand Home Server Module.

trust the HSM and give it permission to execute database requests as if it were the content provider's Web application. In doing so, however, the content provider does not need to trust Ferdinand nodes or modify its security configuration to allow those nodes to directly execute database requests at the home server.

The HSM also contains Encrypter and Decrypter submodules. If a content provider requires database updates, queries, or their results to remain private, the Encrypter and Decrypter can be used to prevent remote Ferdinand nodes from inspecting the database requests and results. See [65] for a more complete discussion of our privacy and security model, implementation, and their performance consequences.

3.2 The Web and application servers

To use the Ferdinand DBSS, the Web and application servers do not need any architectural modifications. Each Web application, however, must be configured to use the Ferdinand database driver rather than the database driver of the home server. For most applications this requires just copying the Ferdinand database driver to a location accessible to the application (e.g. for a Java Web application, to within the Java Virtual Machine's class path) and a one-line source code or configuration file change to use the

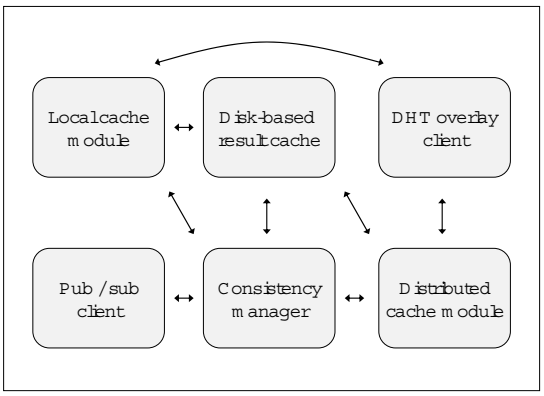


Figure 3.4: Component diagram of a Ferdinand server node.

Ferdinand driver.

Our implementation of the Ferdinand database driver is a lightweight, JDBC driver that simply forwards requests to a Ferdinand node. Like the home server's HSM, the Ferdinand JDBC driver also contains Encrypter and Decrypter submodules. For content providers that require security or privacy guarantees from Ferdinand, the content provider must trust the Ferdinand JDBC driver, which will encrypt requests and decrypt results to prevent their inspection by Ferdinand nodes.

Aside from requiring compatibility with the Ferdinand database driver, Ferdinand does not make any assumptions about the implementation of the Web and application servers. As such, Ferdinand is compatible with a wide variety of overall system architectures: the choice to use Ferdinand is orthogonal to whether the Web and application servers are located at the home server, at a CDN node, or even executed locally by the end-user's Web browser.

3.3 The Ferdinand DBSS node

The design of the Ferdinand DBSS node is the primary contribution of the Ferdinand architecture. Each Ferdinand node is architecturally identical to each other Ferdinand node, containing the same core components. These components are a local cache of database query results, part of a distributed cache storing additional query results in cooperation with other Ferdinand nodes, and a cache consistency manager.

Figure 3.4 diagrams how these components interact. We sometimes refer to the distributed cooperative cache as the *master cache* or *master cache module* since it stores a single master copy of each query result cached by Ferdinand. The Ferdinand DBSS nodes communicate with each other using both a publish / subscribe infrastructure and a DHT overlay.

This section continues by describing the disk-based query result cache in Section 3.3.1. We then describe Ferdinand's distributed query result cache in Section 3.3.2, and its cache consistency manager in Section 3.3.3.

3.3.1 A disk-based cache for database query results

In designing our query result cache, one of our biggest goals was to achieve an efficient, lightweight implementation that avoided complex processing of queries at the DBSS and provide content providers with security and privacy guarantees for their data. To satisfy these constraints, the programming interface for our cache is a simple map between database queries and their query results. This approach has the disadvantage that Ferdinand can only respond to exact matches to previous queries, potentially limiting the hit rate at each cache. It has two key advantages, however: (1) it requires only minimal processing at Ferdinand to respond to a query, and (2) queries and their results can be encrypted such that Ferdinand stores the encrypted query and result in the cache map, enabling it to respond to cache requests without needing to inspect the actual query or its result.

An additional concern was that because each Ferdinand node could provide scalability services for multiple content providers, the overall cache size on each node could be extremely large. This constraint suggests that our cache implementation must be disk-backed or entirely disk-based. We chose to avoid specialized memory-management issues by implementing the disk-backed cache using a commodity relational database: this choice relegates memory management to the cache DBMS implementation of its buffer pool. ‘ The cache consists of a simple *map* relation containing three fields: the database query, the Java hash code of the query, and the cached query result, indexed by the Java hash code. To check if a query Q and its result are present in the cache, the caching module first hashes Q and retrieves all matching entries from the cache database (i.e. *SELECT * FROM map WHERE hash_code = ?*). The

cache module then checks each match against Q, returning the query result if Q was in the cache.

To maximize efficiency and avoid processing of our cache retrieval query, the cache module maintains a pool of connections to the cache database with pre-optimized retrieval queries (i.e. already-constructed Java *PreparedStatement*s). The database statements to add, remove, or update cached results are similarly pre-constructed and pre-optimized.

Our cache size was bounded only such that it would not exceed the available disk space. Cache replacement was implemented using a random replacement algorithm. In practice, none of our experiments encountered the cache size limit and no items were forced from the cache due to storage constraints. A production system might choose a more sophisticated cache replacement policy.

3.3.2 Ferdinand's distributed query result cache

One of the disadvantages of our cache architecture is that each Ferdinand node can only respond to an exact match of a query to which it has already responded, limiting the possible cache hit rate. This limitation is further compounded by our consistency requirements, in which caches are immediately notified of updates and disallowed from responding with a query result it cannot determine to be valid. One of the novel aspects of our approach is that instead of immediately forwarding a cache miss to the home database server, the Ferdinand node instead checks if the query result is present in a distributed query result cache at another Ferdinand node. In our distributed cache design, a query is not forwarded to the home server unless no Ferdinand node has responded to that query since the query was last invalidated by an update. This section describes how we achieve these properties with our distributed query result cache.

Our distributed query result cache has the same interface as the local cache at each node: the distributed cache is also a simple map between database queries and their query results. This interface lends itself to a simple implementation as we can use a commodity distributed hash table to implement the distributed map. It also allows us to use the same disk-based storage module that we use for the local cache at each Ferdinand node. Each query has a *master node* in the DBSS, determined by that query's hash in the DHT. Our caching algorithm enables the master node to cache a query if any Ferdinand node

has executed that query since it was last invalidated.

Our implementation uses Pastry [83], a peer-to-peer routing substrate based on PRR trees [79]. In Pastry, each node is assigned a 128-bit identifier. Each query stored in the Pastry distributed hash table is also associated with a 128-bit identifier of its own, and stored at the node with the closest Pastry node identifier. Pastry provides efficient routing among the nodes, as well as mechanisms to facilitate the persistence and availability of stored objects if a node fails.

We deterministically assign a 128-bit identifier to each stored query using the first 128 bits of the query's SHA-1 digest [70]. Although computation at the DBSS nodes is not a limiting factor in our system design, a simpler, more efficient hash function would perhaps be more appropriate in a deployed Ferdinand system.

To check if a query Q and its result are present in the distributed cache, the distributed caching module computes the SHA-1 digest of Q to determine its 128-bit Pastry identifier. We then use the Pastry network overlay to route a cache request to Q's master node, the Ferdinand node in the Pastry overlay with the identifier closest to Q's. Q's master node then checks its local disk-based cache for Q and its query result, returning the result if present and otherwise forwarding Q to the content provider's home database server. In this latter case, the home database server returns the query result to Q's master node, which places Q and its result in the cache and returns the result to the original Ferdinand node.

As mentioned above, the distributed cache module uses the same disk-based cache at each node as the local cache; Figure 3.5 shows this interaction. This re-use is not just of the code base, but of the cache storage as well. As a side effect of the re-use, a local cache request at a query's master node will result in a local cache hit rather than a master cache hit. This behavior does not affect the correctness of our algorithms, just the accounting of whether the hit is attributed to the local or distributed cache at the master node.

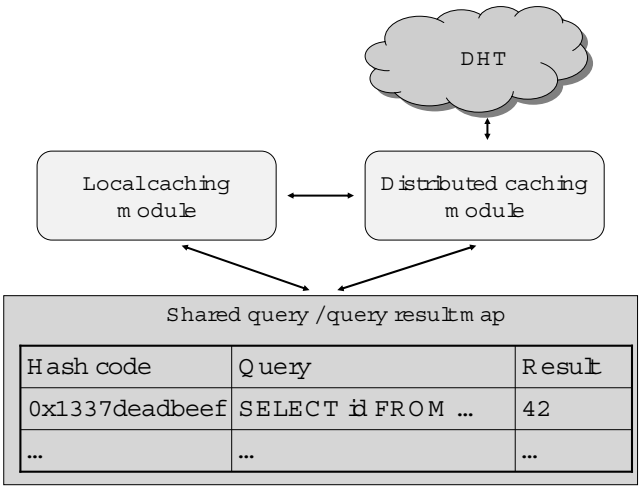


Figure 3.5: Shared architecture of Ferdinand's local and distributed caches. The two caches share physical storage for the query-to-query-result map on each Ferdinand node, so a local cache request can result in a cache hit if the distributed cache is storing the appropriate query result at the local node.

3.3.3 Ferdinand's cache consistency manager

The last major component of Ferdinand's design is its cache consistency manager. In designing the consistency manager we needed to satisfy multiple, somewhat opposing goals. Our first and foremost goal is scalability: the consistency manager must not require additional work from the content provider or the home server database. Second, Ferdinand must offer strong consistency: when possible, it must avoid responding to queries with stale data, and enable the content provider to control the level of consistency. Finally, the consistency manager must be compatible with our dual goal of giving the content provider control over the privacy and security of data, still providing consistency guarantees even when the queries and query results are hidden to Ferdinand.

Our approach is to use publish / subscribe to notify the DBSS nodes of updates without involving the home database server. When a Ferdinand node places a query it its cache, it first subscribes to a set of messages associated with that query. When an update is executed at the home server, Ferdinand publishes messages such that any affected queries will be notified of the update. This approach is highly scalable and stale queries are quickly purged from the Ferdinand caches. Our algorithm maintains full consistency when only single-statement transactions are used, but the consistency is relaxed for

multi-statement transactions and therefore is ill-suited if full multi-statement transactional consistency is strictly required. In these circumstances, the content provider can mark arbitrary queries as uncacheable to ensure that full multi-statement transactional consistency is ensured by the home database server.

Our Ferdinand implementation uses Scribe [84], a topic-based publish / subscribe system, for consistency management. Scribe is completely decentralized and highly scalable, and built upon the routing functionality provided by the Pastry network overlay that we already use for our DHT-based distributed cache. In topic-based publish / subscribe systems, subscriptions specify a fixed topic name, and a subscribed client receives all publications to that topic name until they unsubscribe. Scribe is designed to support a high subscription and unsubscription rate and incurs no load for unused topic names, at the expense of some inefficiency when delivering publication notifications.

A Ferdinand node subscribes to a set of Pastry topics when it caches a database query and broadcasts to some set of topics for each update it issues. To maintain correct consistency, Ferdinand must guarantee that for each update, any proxy caching a database query result affected by that update will receive an update notification. To ensure this property, each cached database query must create a subscription to at least one topic to which that update is published. We call this problem of mapping database queries and updates to publish / subscribe groups the Query / Update Multicast Association (QUMA) problem. We discuss the QUMA problem in greater detail in Chapter 6, where we show how to use an offline analysis of an application and its database requests to define an efficient mapping between queries, updates, and publish / subscribe topics.

In our implementation, for each publish / subscribe topic there is a second *master topic* used for maintaining the consistency of the distributed cache. When a query result is placed in a Ferdinand node's local cache, the consistency manager subscribes to all non-master topics related to that query. When a query result is placed into the distributed cache at the query's master Ferdinand node, the master node subscribes to all master topics related to the query. In Chapter 6 we show that our two-tiered use of publish / subscribe – with separate master and non-master topics – correctly ensures that each Ferdinand node will receive all updates that affect it, while a simpler one-tier implementation would not be sufficient to maintain the consistency of both the distributed master and local non-master caches.

When an update notification is received at a Ferdinand node, we also must map that update to the cached queries it possibly affects. In our implementation, the cache consistency manager explicitly tracks for each topic the queries that might be affected by an update published to that topic. When an update notification is received on a topic, the consistency manager inspects each query associated with that topic, invaliding the query from the cache if the query is possibly affected by the update. In some cases the consistency manager cannot determine if a query is affected or not by an update; in those cases the consistency manager is conservative and invalidates any query which cannot be proven to be unaffected by the update.

3.4 An algorithmic view of the Ferdinand DBSS

Our DBSS architecture uses five main algorithms: (1) for a local database query at a Ferdinand node, (2) for a database query at that query's master node in the distributed cache, (3) for a local database update at a Ferdinand node, (4) for handling an update notification on a master publish / subscribe topic, and (5) for handling an update notification on a non-master publish / subscribe topic. In this section we first describe the external functions and the data objects Ferdinand uses to support these algorithms. We then describe each of these algorithms in turn. Finally, in Section 3.4.7 we briefly discuss how Ferdinand behaves in the presence of node failures.

3.4.1 Functions and data structures used by Ferdinand

Table 3.1 lists the functions used by the Ferdinand algorithms we describe below. These functions can be roughly categorized into three groups: (1) functions provided by the publish / subscribe system, (2) general network communication and functions provided by the DHT overlay, and (3) functions provided by the Ferdinand consistency manager that we describe in Chapter 6.

Table 3.2 lists the data objects maintained and used by Ferdinand's algorithms. The first, the *cacheMap*, implements the Ferdinand cache at each node. The *topicToQueryMap* tracks which queries are possibly affected by publications to each topic to which a Ferdinand node is subscribed, and *pendingAndValid* is

Function	*Inputs*	*Output*
subscribe(t)	t: a pub / sub topic	Subscribes to t.
unsubscribe(t)	t: a pub / sub topic	Unsubscribes from t.
publish(t, obj)	t: a pub / sub topic obj: any object	Publishes obj to t.
dhtSend($dest, obj$)	$dest$: a DHT ID obj: any object	Sends obj to the DHT node that hosts ID $dest$.
homeServerSend(r)	r : a database request	Sends r to the home database server for execution.
waitForReply()		Pauses until the success of the previous command has been confirmed.
computeTopics(r)	r: a database request	Returns the set of non-master pub / sub topics associated with r.
computeMasterTopics(r)	r: a database request	Returns the set of master pub / sub topics associated with r.
nonMasterTopic(t)	t: a master pub / sub topic	Returns the non-master topic that corresponds to t.
possiblyAffects(u, q)	u: a database update q: a database query	Returns *false* if the DBSS can determine that u does not affect q, and *true* otherwise.

Table 3.1: External functions used by Ferdinand's algorithms described here.

Object name	*Description*
cacheMap	A map between database queries and their query results. Supports addition and removal of query-result pairs, and retrieval of a query's result in the map.
topicToQueryMap	A map between pub / sub topics and the set of queries possibly affected by publications to the topic. Supports addition and removal of topic-query pairs, and retrieval of a topic's set of possibly affected queries.
pendingAndValid	The set of pending queries at a Ferdinand node that are still valid. Supports addition and removal of queries to the set, and lookup of whether a query is in the set.

Table 3.2: Data objects used by the Ferdinand algorithms described here.

a set of queries used to mark which pending queries can be safely placed in the cache when their results are obtained.

We continue by describing the various Ferdinand algorithms that maintain the Ferdinand cache and respond to database requests. The descriptions here are slightly simplified from our actual implementation, as our implementation contains significant error checking as well as synchronization code to ensure that local concurrent processing cannot lead to an invalid state. (As an example of such synchronization, a database query will not be locally executed while another instance of the same query is pending.)

3.4.2 Processing a local query at a Ferdinand node

Algorithm 3.1 shows the algorithm for processing a database query at a local Ferdinand node. When a query is first received by Ferdinand at some node, the node first checks its local cache for the query result. If the query result is present in the local cache, the node immediately replies with the query result, completing the interaction. If the query result is not present in the cache, the consistency manager computes any publish / subscribe topics associated with the query, subscribes to any topics to which it

Algorithm 3.1 Processing a local query at a Ferdinand node. As input, this algorithm takes a database query Q.

```
01  R ← cacheMap.get(Q)
02  if (R) return R
03  pendingAndValid.add(Q)
04  topics ←computeTopics(Q)
05  foreach t ∈ topics
06      topicToQueryMap.add(t, Q)
07      subscribe(t)
08      waitForReply()
09  R ←dhtSend(hash(Q), Q)
10  if pendingAndValid.contains(Q)
11      cacheMap.add(Q, R)
12      pendingAndValid.remove(Q)
13  return R
```

Algorithm 3.2 Processing a query at its master Ferdinand node. As input, this algorithm takes a database query Q.

```
01  R ← cacheMap.get(Q)
02  if (R) return R
03  pendingAndValid.add(Q)
04  topics ←computeMasterTopics(Q)
05  foreach t ∈ topics
06      topicToQueryMap.add(t, Q)
07      subscribe(t)
08      waitForReply()
09  R ←homeServerSend(Q)
10  if pendingAndValid.contains(Q)
11      cacheMap.add(Q, R)
12      pendingAndValid.remove(Q)
13  return R
```

is not already subscribed, and waits for confirmation of each subscription. For each subscription, the consistency manager tracks which queries are associated with that subscription. Once all subscriptions are confirmed, the Ferdinand node hashes the query and forwards the request to the query's master node in the distributed cache. When the query result is returned to the local node, the node checks if any intervening updates might have invalidated the query since the query was forwarded to the master node. If no updates have affected the query, the query result is added to the local cache. Finally, the query result is returned.

Algorithm 3.3 Processing an update at a Ferdinand node. As input, this algorithm takes a database update U.

```
01  R ←homeServerSend(U)
02  waitForReply()
03  topics ←computeMasterTopics(U)
04  foreach t ∈ topics
05        publish(t,U)
06  return R
```

3.4.3 Processing a query at its master Ferdinand node

Algorithm 3.2 shows the algorithm for processing a database query at its master Ferdinand node. When a query is received by its master node in the DHT overlay, the master node first checks for the query result in its local cache. If the query result is present in the cache, the master node immediately replies with the result, completing the interaction. Otherwise, the master node proceeds much like the local node: it computes all master (rather than non-master) topics associated with the query, subscribes to those topics and waits for confirmation, and forwards the query to the home database server. When the query result is received it likewise checks for intervening updates and places the query result in its cache if possible. It finally returns the query result to the original Ferdinand node where the query was issued.

3.4.4 Processing an update at a Ferdinand node

Algorithm 3.3 shows the algorithm for processing a database update at a Ferdinand node. When an update is received at a Ferdinand node, it is immediately forwarded to the home database server. When the home database server replies, the consistency manager computes all master publish / subscribe topics associated with that update and publishes the update to those topics. The Ferdinand node finally replies with confirmation of the update.

Algorithm 3.4 Handling an update notification on a master topic. As input, this algorithm takes the published update U and the topic T on which the update notification was received.

```
01  queries ← topicToQueryMap.get(T)
02  foreach q ∈ queries
03      if possiblyAffects(U, q)
04          pendingAndValid.remove(q)
05          cacheMap.remove(q)
06          reducedTopics ←computeMasterTopics(q)
07          foreach t ∈ reducedTopics
08              topicToQueryMap.remove(t, q)
09              if (topicToQueryMap.get(t) = ∅)
10                  unsubscribe(t)
11  publish(nonMasterTopic(T), U)
```

3.4.5 Handling an update notification on a master topic

Algorithm 3.4 shows the algorithm for processing an update notification on a master publish / subscribe topic. The consistency manager first computes any cached queries that are associated with the master topic, by looking up these queries in its topic-to-query map. For each possibly-affected query, the node applies simple query-update independence analysis to check if the query result is possibly affected by the update. If the query and update cannot be determined to be independent, the query result is removed from the cache and removed from the topic-to-query map. In the case where the invalidated query is pending, the node simply marks the pending query as invalidated to prevent its query result from being cached when the result is received. The Ferdinand node unsubscribes from any master topics on which no cached queries depend, and finally republishes the update on the non-master topic that corresponds to the master topic on which the notification was received.

Algorithm 3.5 Handling an update notification on a non-master topic. As input, this algorithm takes the published update U and the topic T on which the update notification was received.

```
01  queries ← topicToQueryMap.get(T)
02  foreach q ∈ queries
03      if possiblyAffects(U, q)
04          pendingAndValid.remove(q)
05          cacheMap.remove(q)
06          reducedTopics ←computeTopics(q)
07          foreach t ∈ reducedTopics
08              topicToQueryMap.remove(t, q)
09              if (topicToQueryMap.get(t) = ∅)
10                  unsubscribe(t)
```

3.4.6 Handling an update notification on a non-master topic

Algorithm 3.5 shows the algorithm for processing an update notification on a non-master publish / subscribe topic. The algorithm is very similar to that for processing an update on a master topic. The Ferdinand node likewise uses its topic-to-query map to determine any possibly-affected queries and removes query results that it cannot determine to be unaffected by the update. It similarly marks any affected pending queries as invalid and unsubscribes to any non-master topics on which no cached queries depend.

3.4.7 Ferdinand's behavior with node failures

Nodes occasionally fail in any system, and a DBSS must continue to operate if a DBSS node fails. For the most part, Ferdinand does not extensively provision for node failures and instead relies on the fault tolerance of its DHT and publish / subscribe components. If the publish / subscribe system reliably

operates in the presence of node failures, then nodes that have not failed should continue to receive update notifications for their local caches much as if there was no node failure.

DHTs often respond to node failures by shifting the objects from the failed node to other DHT nodes, using backup copies as the source of objects on the failed node. In our design, this solution does not easily apply, because the Ferdinand nodes assigned to cache a query result from a failed node must also subscribe to consistency messages for that query result. There are two possible ways to address this. First, some DHT implementations warn application clients (i.e., Ferdinand) when objects are shifted between DHT nodes. In this case, a Ferdinand node could subscribe to the topics necessary to maintain the consistency of its newly assigned query results. If the DHT does not provide this warning, then our second solution applies: Ferdinand can simply invalidate all objects in the cooperative cache. This would result in a temporary performance degradation as Ferdinand would lose all objects in the cooperative cache, but local caches could still be used, and Ferdinand's performance would gradually recover as the cooperative cache was repopulated.

When a failed Ferdinand node rejoins the system, it rejoins the DHT network and publish / subscribe system, and invalidates all stored query results from before its failure. Note that if the publish / subscribe system queues messages for a failed node and delivers them when the node returns, then a recovered Ferdinand node does not need to invalidate the query results from its local cache.

Chapter 4

On the Evaluation of a DBSS

In this chapter, we describe at a high level our methodology for evaluating a Database Scalability Service. We first describe the relevant metrics of a DBSS's performance and the consequences of those metrics in Section 4.1. Section 4.2 describes the benchmark applications we use to examine the DBSS's performance. Section 4.3 describes some side effects of our experimental methodology on performance measurements and justifies that our methodology accurately represents the performance of a DBSS, and Section 4.4 briefly discusses the environment we use to evaluate our DBSS implementation.

4.1 Performance metrics for a DBSS

Constructing a sound performance metric for a DBSS is not as trivial as it seems. First, there are multiple metrics by which to measure the performance of the DBSS. There is also a question of what should be the scope of these measurements: should we measure just the internal performance of the DBSS (e.g., the database throughput), or measure the performance of the overall system that uses the DBSS to generate dynamic Web content (e.g., the Web throughput)? In this section we discuss these issues in greater detail and outline our approach in evaluating the DBSS.

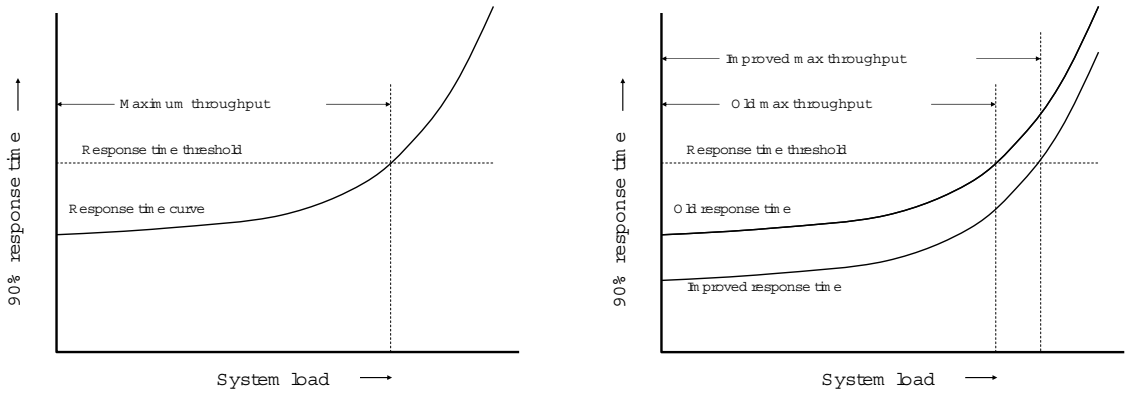
The two most important performance metrics are the maximum throughput that can be sustained by the DBSS, and the latency of responding to a request. In some cases, however, achieving high throughput

and low latency are conflicting goals. One way to maximize throughput might be to saturate the system with requests; such a strategy might enable the DBSS to execute the requests more efficiently on average, but incur long delays for each interaction since so many requests are outstanding at each point in time. Another possible scalability metric is the number of users or connections simultaneously supported by the DBSS. Again this goal can conflict with the latency goal. Finally, if overall scalability is not affected, we prefer to minimize the internal resources used by the DBSS. Thus, we have the secondary goal of reducing DBSS disk storage, CPU load, and network traffic between DBSS nodes.

One resolution to the competing goals of throughput and latency is to only consider the throughput of system configurations that meet a given latency threshold. This is the approach taken by the Transaction Processing Council in formulating their standards of measurement for the TPC-W benchmark [95]: they require that 90% of responses are generated within a given latency, and that the mean latency is also below a certain threshold. This approach blends the metrics of throughput and latency, creating an overall scalability metric that reflects the scalability of the system while ensuring that each user's experience meets a quality-of-service requirement.

The second question is of the scope of measurement: should we measure the internal performance of the DBSS (i.e. the throughput and latency of database requests), or of the overall system to generate Web content? The former approach yields more direct measurements of the DBSS, but those measurements are of internal properties only visible to the content provider. The latter approach is less direct but also less artificial, measuring the throughput and latency of requests to the end user.

Overall, our focus is to guarantee good performance to the end user, while meeting the content provider's scalability needs. For this purpose, measuring the performance of the overall system is more ideal than just measuring the database component. Our main metric therefore is the throughput of end user Web requests, while responding to those requests within a latency threshold specific for each Web application. For systems that provide equal scalability we additionally apply our secondary metrics, attempting to minimize DBSS disk storage, CPU load, and network communication.



(a) Typical response time for a DBSS workload

(b) Improved throughput as a consequence of improved response time

Figure 4.1: Relationship between scalability and response time as a consequence of our throughput metric. If the DBSS can improve response time while performing the same amount of work, overall scalability slightly improves since an improved maximum throughput can be achieved without exceeding the response time thresholds.

4.1.1 Effect of response time on scalability

Because our scalability metric takes into account response time as well as the rate at which the DBSS responds to requests, the overall scalability of the DBSS is intricately linked to both of these performance properties. Because of this, design choices that affect DBSS response times can also affect scalability, even when the DBSS is performing the same overall amount of work.

To clarify the relationship between scalability and response time, consider the example performance measurements of Figure 4.1a. This graph depicts the typical relationship between the 90th percentile response time and system load for a system that generates dynamic Web content. As the system load increases and throughput increases, the time needed to generate a response usually increases as well. Using our scalability metric, the maximum throughput of the system is constrained by the system's ability to respond to requests fast enough to satisfy the response time threshold.

Consider an improved architecture that could somehow respond to requests faster by some constant time, but otherwise did the same amount of work, used the same resources, and exhibited the same per-

formance characteristics. Figure 4.1b depicts the performance that might result along with the performance of the original system. Even though the improved architecture has the same resource bottlenecks and does the same amount of work, its lower response times enable it to satisfy the response time threshold at a higher system load than the unimproved system. Thus, by our scalability metric the improved system is more scalable.

The relationship between scalability and response time latency therefore has interesting consequences for our system design choices. System designs that reduce the use of bottleneck resources and improve the efficiency of dynamic Web content generation might actually reduce scalability, if those designs incur delays that increase typical response times. This limitation is particularly relevant in high-latency network environments where communication among the DBSS nodes and the home server can cause a significant delay. In these environments, overall scalability can potentially be limited by the network latencies and communication requirements, not just by more typical resource constraints such as network bandwidth, memory, CPU load, and disk throughput.

4.2 Benchmark applications

Having chosen to evaluate the DBSS as part of a larger system to generate dynamic Web content, we use data intensive Web applications designed to exercise and evaluate such a system. There are three commonly-used benchmark applications designed for this purpose, and we use all three. Each of these applications is designed to simulate a dynamic online Web site: the TPC-W bookstore benchmark, the RUBiS auction site, and the RUBBoS bulletin board system. This section continues by discussing the properties common to all three of these benchmarks, and then discusses each benchmark in greater detail.

4.2.1 Common properties of the benchmark applications

All three benchmarks are implemented as a collection of Java programs, with each program generating a single page of the dynamic Web site. Like many Web applications, all of the benchmarks generate database requests using database templates, enabling us to analyze their database workloads offline. The

Application	Programs	Query templates	Update templates	Database tables
TPC-W bookstore	14	28	16	10
RUBiS auction	20	28	11	8
RUBBoS bulletin board	18	38	13	8

Table 4.1: Static properties of the benchmark Web applications

auction and bulletin board benchmarks also include a small number of static Web pages which are not generated dynamically with a Web application.

Table 4.1 presents rudimentary properties of each of the benchmark applications. Most of the Java programs maintain a pool of connections to the database and issue one or more database requests during execution. Overall, each of the benchmarks contain a greater number of query templates than update templates. At runtime the benchmarks are even more read-oriented than suggested by the static properties: on average, the programs execute many more database queries than updates. This property is considered typical of many dynamic Web applications.

During execution, each benchmark simulates the activity of a collection of end users as they browse the online site. Each user is referred to as an *emulated browser*. An emulated browser sends a request to the Web server, waits for the response, and then "thinks" for a moment before sending another request. Eventually each emulated browser concludes its session and another emulated browser is simulated by the benchmark. In all of our experiments we use the standard think and session times recommended by the Transaction Processing Council. For think time, this is an exponential distribution around a mean of 7 seconds, and for session time an exponential distribution around a mean of 15 minutes.

4.2.2 The TPC-W bookstore benchmark

The TPC-W bookstore benchmark [95] emulates the functionality of a simple online bookstore, in the spirit of Amazon.com [7]. Typical user interactions include browsing the bookstore, registering user accounts, adding or removing books from their shopping cart, completing purchases, or searching the

	Admin confirm	Admin request	Best sellers	Buy confirm	Buy request	Customer registration	Home	New products	Order display	Order inquiry	Product detail	Search request	Search results	Shopping cart
90% threshold response time (s)	20	3	5	5	3	3	3	5	3	3	3	3	10	3

Table 4.2: Threshold response times for the TPC-W Web interactions. For each interaction type, 90% of the responses for that type must be generated within the number of seconds shown above. The mean response time for that interaction type also must not exceed the same threshold.

bookstore inventory by any of several criteria like author, book subject, or popularity. Overall, the bookstore benchmark results in a more balanced database workload than the auction or bulletin board benchmarks. No table in the TPC-W database dominates the query or update load. Our version of the TPC-W database contains 1 million items for sale, with a total database size of 480 MB.

The TPC-W bookstore can be configured to generate three distinct workloads. The main workload, the *shopping mix* is designed to emulate the most common workload at an online bookstore, with approximately 80% of the customer interactions be simply browsing the site and 20% of the interactions resulting in purchases. The *browsing mix* is an even more read-oriented workload, with 95% browsing and only 5% purchases. The third workload, the *ordering mix* is much more update-oriented, with a 50% split between browsing and purchasing.

We use only the first two workloads, the shopping and browsing mixes. The 50% ordering mix is considered an oddity by most researchers and rarely used to evaluate experimental systems. For the bookstore, the run time proportion of queries and updates is slightly more read-oriented than the workload interaction mix suggests: the shopping mix results in a mix of about 85% database queries and 15% updates, while the run time proportion is close to 97% queries and 3% updates for the browsing mix.

The throughput of the TPC-W benchmark is defined as the maximum throughput obtained while re-

sponding to 90% of each type of interaction under a given response-time threshold. Table 4.2 lists the various TPC-W interaction types and their response-time thresholds, as given by the TPC-W specification [95]. For most interactions the 90% threshold is three seconds; for some complex interactions the threshold is five or ten seconds, and the threshold is twenty seconds for a rare administrative interaction.

Our TPC-W implementation contains one major modification from the standard specification. In the original benchmark all books are uniformly popular. Our version uses a Zipf distribution of book popularity, which Brynjolfsson et al. showed to be more realistic of online booksellers in [23]. Specifically, we model book popularity as $\log Q = 10.526 - 0.871 \log R$ where R is the sales rank of a book and Q is the number of copies sold in a short period of time. Our modified version has the advantage that our experimental results more accurately predict our system's performance on real workloads. However, using a modified implementation does have the disadvantage that our results are not directly comparable to those of other experimental systems that use the unmodified benchmark implementation.

4.2.3 The RUBiS auction benchmark

The RUBiS auction benchmark [71] emulates the functionality of an online auction site modeled after eBay [36]. Typical user interactions include browsing current auctions, bidding on current items, placing items for sale, registering for the site, or commenting on previous transactions. The auction benchmark results in a less balanced workload than the bookstore benchmark: the `items` table on the auction site receives the majority of queries and updates, and thus is a hot spot in the auction database. Our version of the RUBiS database contains 33,667 auction items and 100,000 registered users, for a total database size of about 990 MB.

RUBiS is configurable to use two distinct workload mixes. The *bidding mix* is the primary workload, consisting of 85% read-only interactions and 15% of Web interactions involving an update. We do not use RUBiS's second workload, a *browsing mix* made of only read-only interactions. Even though 85% of the bidding mix Web interactions contain updates, the database workload is more read-oriented than this implies: during normal execution, the database workload is about 93% database queries and only 7% updates.

A typical Web interaction in the auction benchmark requires more computation than for the bookstore. The database workload is extremely read-oriented, but many auction queries embed the current time (e.g. "Show all auctions that end within an hour of now") and thus can never result in a cache hit. Note that an application designer who was aware that the auction site might be used on a caching architecture like our DBSS might improve cache performance by rewriting these time-based queries such that they are not all distinct.

The RUBiS specification does not declare 90% threshold latencies for each interaction like the TPC-W specification does. For experiments involving the RUBiS auction benchmark, however, we chose to define throughput similarly, as the maximum throughput achieved while ensuring that 90% of interactions were completed within 3 seconds. A notable difference between this definition and that of TPC-W is that the RUBiS 90% threshold is for *all* interactions, not a separate threshold for each interaction type. This allows us to use a practical definition of throughput similar to the definition for TPC-W, without having to determine an appropriate latency threshold for each interaction type. Using a single threshold for all interactions, however, does mean that a rare interaction type (totaling less than 10% of all) could require arbitrarily high latencies without invalidating the experimental results. In practice, however, we determined that even though we did not set a separate latency threshold for each interaction type, even the slowest rare interaction types were typically completed in five to ten seconds.

4.2.4 The RUBBoS bulletin board benchmark

Our final Web application benchmark is the RUBBoS bulletin board benchmark [72], modeled after the Slashdot news and bulletin board site. It allows users to undertake actions such as register for and browse the Web site, submit new articles, comment on existing articles, or moderate comments. In the RUBBoS workload the majority of database requests access the `stories`, `comments`, or `users` tables. Our version of the RUBBoS database contains 6,000 active stories, 60,000 old stories, 213,292 comments, and 500,000 users, with a total database size of about 1.6 GB.

The RUBBoS bulletin board contains three distinct workload mixes: a *browse-only mix*, a *user mix*, and an *author mix*. The browse-only mix consists of only read-only interactions, with no updates to the

database. The interactions of the user mix are about 70% read-only, and 30% containing updates. The author mix contains a 50% mix of update and read-only interactions. Of these three we use only the user mix, with its 70-30 ratio: we feel that both the 50-50 mix and the read-only workload are unrealistic for a typical Web application.

Some of the bulletin board programs execute many queries for a single interaction. Even using the user mix, 98% of all database requests are queries, with only 2% being updates. We use the same throughput definition as with the auction benchmark: the maximum throughput is that achieved while responding to 90% of all interactions within three seconds.

4.3 Workload generation in our evaluation model

One difficulty in evaluating the performance of a system is that seemingly minor considerations can significantly affect the system's performance. This section describes one such consideration for systems with work load queues: the effect of the methodology of workload generation on a system's performance. We start by discussing different methods to generate workloads, discuss the difference between our experimental workloads and the workloads a DBSS might encounter as a deployed system, and finally justify that this difference does not invalidate our performance results.

In [86], Schroeder et al. describe how the methodology of workload generation affects mean response time in queue-based systems, extending upon the earlier work of Bondi and Whitt [21]. In particular, Schroeder et al. describe two common methods of workload generation, called *open* or *closed* systems. In an open system, requests are introduced to the system based on external factors; often, a new request is issued at a fixed interval regardless of current system performance. In contrast, a closed system contains a fixed number of workload generators that each issue a request and wait for a response before issuing additional requests.

In a queue-based system there are two key differences between an open and closed evaluative framework. First, the overall workload in a closed system is partially determined by the system performance. As the mean response time increases along with the system load, the time between new requests also

increases since new requests are not issued until the workload generator receives an old request's response. In other words, as the system load increases, the request rate *decreases*, which can have a stabilizing effect on system performance.

Second, in a closed system the work queue is always a fixed length, because there are always the same number of outstanding requests, the number of workload generators. In contrast, an open system can have an unbounded queue length. When the overall request rate is small compared to the potential throughput of the system, the work queue will also be small. But if the request rate exceeds the maximum system throughput, new requests will continue to enter the system at their normal rate even at high system loads and the work queue can grow without bound as the request rate remains high. In an open system, the mean response time can then grow arbitrarily long for even a fixed workload as new requests spend an increasingly long time in the work queue. As Schroeder et al. discuss, the work scheduling algorithm becomes increasingly important for an open system since the effect of scheduling in a closed system is constrained by the bounded queue length: even for equivalent overall system loads, a poor job scheduling algorithm on a long queue can significantly increase the mean response time of a system.

For first-come first-served scheduling of any workload, it is known that the response time of an open system is an upper bound for the response time of the closed system [85]. Schroeder et al. show that for high system loads there can be a great difference in the response times of open or closed systems [86]. However, they also show that this difference is much less pronounced for low system loads when the queue of even the open system is expected to be short. They also show that the difference between open and closed systems can be further mitigated by using a high multiprogramming level (i.e., a large number of workload generators) in the closed system, because the long queue of a closed system with a high multiprogramming level more closely mimics the queue of an open system under high load. Finally, the difference is greatly reduced when the work required to generate a response is similar for most requests.

All of our benchmark Web applications are closed systems, with new requests only generated after a response has been received. For real Web applications, however, the workload more closely mimics what Schroeder et al. call a *partly-open system*, where new requests are generated by a mix of new

users entering the system and old users who only issue a request after a previous response is received. Thus, we need to justify that our experimental results from a closed system are valid for the partly-open systems that we expect to encounter in the real world. For our results to be valid, we must ensure that our closed system is used for only low system loads, the multiprogramming level is high, or that most requests require similar processing time.

Because our goal is to measure peak throughput of various experimental systems, we do not have the option of measuring performance at only low system loads. Our workloads do give us two main tools for controlling the overall request rate: we can directly change the multiprogramming level (number of emulated browsers) or the think time between responses and new requests. Using a fixed TPC-W standard think time of 7 seconds, our peak throughputs were often reached with a very high number of simultaneous emulated browsers, usually greater than a thousand. For this multiprogramming level and the system loads at which maximum throughputs are reached while meeting latency requirements, the differences are slight between open and closed systems as studied by Schroeder et al. Thus, the difference is similarly slight between our closed experimental system and the partly-open workload of real Web applications, and our overall results should be valid.

4.4 Our experiment environment

For this thesis we conducted experiments in network environments with varied properties. To give us adequate control over the network environment, we conducted most experiments on the Emulab network testbed [98]. The Emulab testbed gives the experimenter full control over experiment nodes for a limited time, and can emulate restricted network connectivity and loss between the experiment nodes.

For all of our Emulab experiments we configured the home database server to run on a 3 GHz Intel Pentium Xeon server with 1 GB of memory and a large 10,000 RPM SCSI disk. For most experiments the DBSS nodes run on similar hardware. In most of our experiments we configured the application and Web server to run on the same computer as the DBSS node, mimicking the overall architecture where the DBSS is either co-located with a content provider or hosted by a CDN. The benchmark client

browsers were emulated on separate 850 MHz servers, directly connected to the DBSS node via a high speed point-to-point link. In various experiments we use the Emulab facilities to restrict the bandwidth and increase the latency among DBSS servers and the home database server to emulate the particular network environments for different architectures that use the DBSS.

We also conducted a small number of experiments using a custom simulator for the DBSS cache and network. The simulator allows us to examine DBSS configurations that are too large to practically test in a real or emulated environment. We describe the simulator in more detail in Chapter 6.

4.4.1 Implementation details for our DBSS prototype

In our Ferdinand prototype all DBSS components are implemented in Java 1.5.0 and run using the Sun Microsystems Java 1.5.0-12 Runtime Environment. This includes the Home Server Module, the main program at each Ferdinand node, and the Ferdinand JDBC driver implementation used by the Web applications within the application server.

The home database server runs MySQL 4 and is configured to use most of the available memory for its buffer pool, but to enable internal local caching with a 32 MB cache size. The Home Server Module connects to the home server MySQL database using the MySQL Connector/J JDBC driver.

The DBSS nodes also run MySQL 4 for its disk-based cache, also connecting with the MySQL Connector/J JDBC driver. To route messages for the distributed cache, Ferdinand uses FreePastry 2.0_04, an implementation of the Pastry [83] peer-to-peer substrate. For consistency management the DBSS nodes use the Scribe [84] publish / subscribe system based on Pastry, also from the FreePastry 2.0_14 package.

For our Web and application servers we use Apache Tomcat 4.0.4, configured in standalone mode as a cache for static content and as a servlet container.

Chapter 5

The Scalability of the Ferdinand DBSS

This chapter presents our evaluation of the Ferdinand Database Scalability Service. Our main goal was to determine the feasibility of the Ferdinand architecture: whether distributed query result caching and publish / subscribe for consistency management can be used to create an effective DBSS. To evaluate the contribution of distributed query result caching, we implement a Ferdinand-like DBSS, called SIMPLE-CACHE, where each DBSS node contains a local query result cache but no shared, distributed cache. We also compare Ferdinand to a traditional three-tiered home server architecture, as well as to a CDN-like system that scales the Web and application servers but does not perform any database caching.

This chapter continues by describing the architecture of three competing approaches to Ferdinand in Section 5.1. Section 5.2 then discusses how we chose several of our experimental parameters, namely how we initialized the cache state and the size of the DBSS for each workload. Section 5.3 presents the scalability measurements for Ferdinand and its competing systems in a typical high-bandwidth, low-latency network environment. Section 5.4 describes the performance of Ferdinand in higher-latency environments, such as a DBSS might encounter if distributed among nodes in a wide-area network like the Internet. Section 5.5 discusses the conclusions we draw from our evaluation of Ferdinand.

5.1 Alternative approaches to Ferdinand

This section describes three alternative approaches to Ferdinand. The first is SIMPLECACHE, a 1-tier caching system much like Ferdinand. The second is a CDN-like system, NOCACHE, and the third is the traditional three-tier architecture, NOPROXY. We continue by discussing each in turn.

5.1.1 The architecture of the SIMPLECACHE DBSS

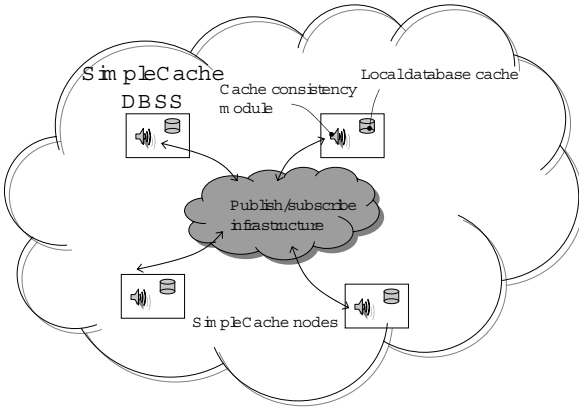


Figure 5.1: High-level architecture of the SIMPLECACHE DBSS. SIMPLECACHE provides local caching at each DBSS node, but unlike Ferdinand does not use a distributed query result cache.

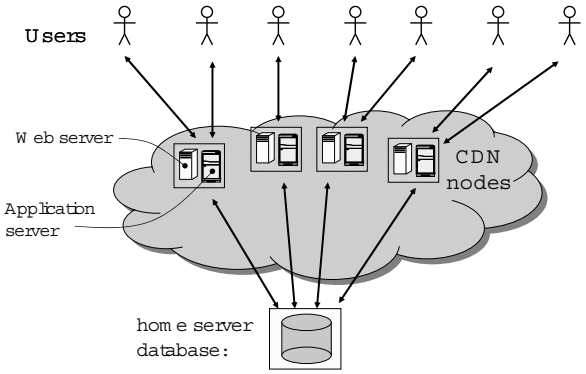


Figure 5.2: High-level architecture of a CDN-like scalability service. The CDN-like service provides no database scalability, so the home server database will quickly become a performance bottleneck.

The design of the SIMPLECACHE DBSS is very similar to that of Ferdinand. SIMPLECACHE's architecture is depicted in Figure 5.1. The key difference between the two implementations is that SIMPLECACHE does not contain a distributed query result cache or the infrastructure needed to support it. When a query result is not present in the local cache, the query is immediately forwarded to the home database server rather than to another DBSS node. Updates are forwarded directly to the home server, just as with Ferdinand.

Like Ferdinand, SIMPLECACHE uses publish / subscribe for consistency management. When a query

is placed in the cache at a SIMPLECACHE node, the node subscribes to the same set of topics to which it would subscribe if it were a local cache in the Ferdinand DBSS. Unlike Ferdinand, SIMPLECACHE does not require a tiered publish / subscribe implementation: with no master result caches for each query, there is no need for separate subscriptions to the master and non-master topics as with Ferdinand. In Chapter 6 we discuss alternative ways to configure Ferdinand's publish / subscribe consistency management system. In all of our experiments, both Ferdinand and SIMPLECACHE use the Topic-by-update configuration we describe there.

Our implementation of SIMPLECACHE uses three main algorithms: (1) for processing a database query at the local DBSS node, (2) for processing an update at the local DBSS node, and (3) for handling an update notification. The query and notification-handling algorithms are very similar to their respective master node algorithms in Ferdinand, and the update processing algorithm is very similar to Ferdinand's update processing. SIMPLECACHE uses many of the same external functions and data objects as Ferdinand, with the exception of the DHT-related functions. For completeness, these algorithms are described in more detail in Appendix A.

Compared to SIMPLECACHE, Ferdinand has the advantage of offloading additional queries from the central database; each database query needs to be executed at the central database only once between any updates that invalidate it. Ferdinand's distributed caching algorithm, however, increases the latency cost of an overall cache miss since the database query is first forwarded to its master DBSS node before it is executed at the home server database. If the latency between DBSS nodes is high, the cost of forwarding a query to its master can be prohibitive, especially for Web applications that execute many database queries for each Web interaction. Distributed query caching also adds additional complexity to the cache consistency mechanism, with Ferdinand requiring twice as many publish / subscribe topics and an extra communication step per update compared to SIMPLECACHE. We quantitatively examine the costs and benefits of distributed caching compared to simple caching in Section 5.3.

5.1.2 A non-caching CDN-like scalability service

To evaluate the effectiveness of database query result caching in general, we also compare the Ferdinand and SIMPLECACHE DBSSes to a CDN-like non-caching architecture. In the CDN-like system, which we call NOCACHE, the Web and application servers are replicated at scalability service nodes, but no attempt is made to scale the database component. All database requests are forwarded directly to the content provider's home database server. Figure 5.2 shows this architecture, which closely resembles that of the Akamai EdgeJava system [3]. While the NOCACHE architecture makes no attempt to scale the database component, its overall design is much simpler since no consistency management is needed at the scalability service nodes; the scalability service nodes also do not require the large disk storage for the database query result caches of Ferdinand and SIMPLECACHE.

5.1.3 A central home server without proxy servers

Finally, we compare Ferdinand against the traditional three-tiered architecture, a central home server that runs the Web server, application server, and database server locally. This architecture was previously depicted in Figure 1.1, and we call it NOPROXY.

5.2 Two DBSS experimental parameters

This section discusses how we chose values for two of Ferdinand's and SIMPLECACHE's major experimental parameters. Section 5.2.1 discusses how we initialized the cache state for both SIMPLECACHE and Ferdinand, and Section 5.2.2 discusses how we chose the size of the DBSS for each workload.

5.2.1 Initializing the SIMPLECACHE and Ferdinand caches

Since we expect the backend database to be the performance bottleneck for the DBSS configurations we test, we expect that system throughput will improve as the cache warms, and measure peak performance

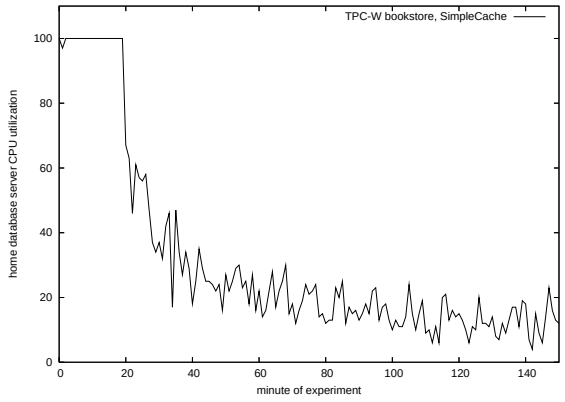


Figure 5.3: CPU utilization of the home database server as the cache warms.

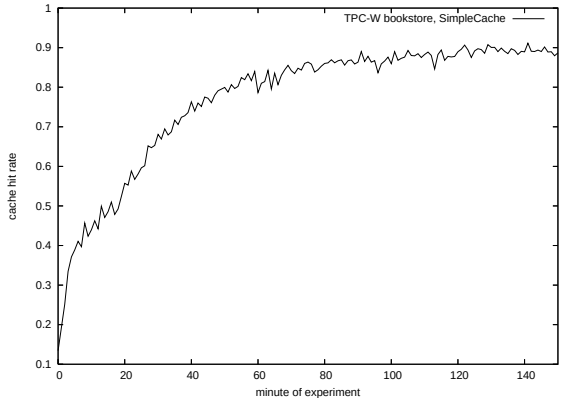


Figure 5.4: Minute-by-minute hit rate of a SIMPLECACHE cache as it warms.

using a warmed initial state. Since the cache behavior differs for the SIMPLECACHE and Ferdinand DBSSes, however, we cannot sensibly initialize the different systems to the same warm state. For our scalability experiments to accurately represent the real throughput of each system, we began by evaluating for how long a SIMPLECACHE and Ferdinand cache must be warmed before it reaches its steady state, when the the cache hit rate does not significantly improve further.

Figure 5.3 shows the CPU utilization of the home database server for each minute of execution as the system warms, for a one-node SIMPLECACHE implementation starting with an empty initial cache. The workload here was generated by the TPC-W bookstore benchmark at a constant system load such that the home database server was slightly overloaded when the cache was empty. At first, the database server runs at 100% CPU utilization as it is unable to respond to requests at the pace that they are sent to the home server. As the cache hit rate improves, however, the home database server eventually clears its request queue and quickly becomes lightly loaded, needing to respond only to the new requests that miss in the DBSS cache. This experiment highlights the effectiveness of query result caching to shield the home server database from queries, and confirms that the home database server will be the performance bottleneck when the cache hit rate is low even for small DBSS implementations.

Figure 5.4 shows the cache hit rate for the same experiment, also as a function of the number of minutes of execution. The cache quickly warms to reach a high fraction of its steady state hit rate,

and thus we expect that the DBSS would achieve near-peak throughput quickly in a real environment. However, the cache hit rate continues to slightly improve as the experiment progresses for a long time. We performed similar experiments for the other benchmark applications and for Ferdinand, with similar results.

Based on these results, for our scalability experiments we chose to warm our caches for a very long time – about six hours each – to reach as close to the actual steady state behavior as possible. We warmed the Ferdinand and SIMPLECACHE caches for equal amounts of time for each workload.

5.2.2 Choosing an appropriate DBSS size

One additional parameter in designing a DBSS scalability experiment is the number of DBSS nodes to use in the system. If the number of DBSS nodes is too small, then the scalability service itself might be the limiting factor in system performance. On the other hand, if the DBSS uses too many nodes for an application, this might reduce the hit rate at any particular cache since each node would then encounter a smaller fraction of the query workload. This problem would be particularly restrictive for the SIMPLECACHE DBSS since additional cache misses would be forwarded directly to the home database server instead of to a distributed cache as in Ferdinand. Thus, using a very large DBSS might actually reduce its maximum throughput for a given application. Our initial experiments confirm that increasing the DBSS size can reduce the cache hit rate at the individual DBSS nodes for the SIMPLECACHE system.

To avoid unfairly disadvantaging SIMPLECACHE in its comparison to Ferdinand, we therefore chose to use relatively small DBSS configurations, but large enough such that the DBSS itself was not the performance bottleneck. For each workload we used a warm initial cache as described above, and tested various DBSS configurations (at increments of 2 nodes per configuration) to determine the best DBSS size for that workload. For the TPC-W bookstore browsing mix, this was a DBSS configuration of 12 DBSS nodes. For the TPC-W bookstore shopping mix, the RUBiS auction, and the RUBBoS bulletin board, the best configuration used only 8 DBSS nodes.

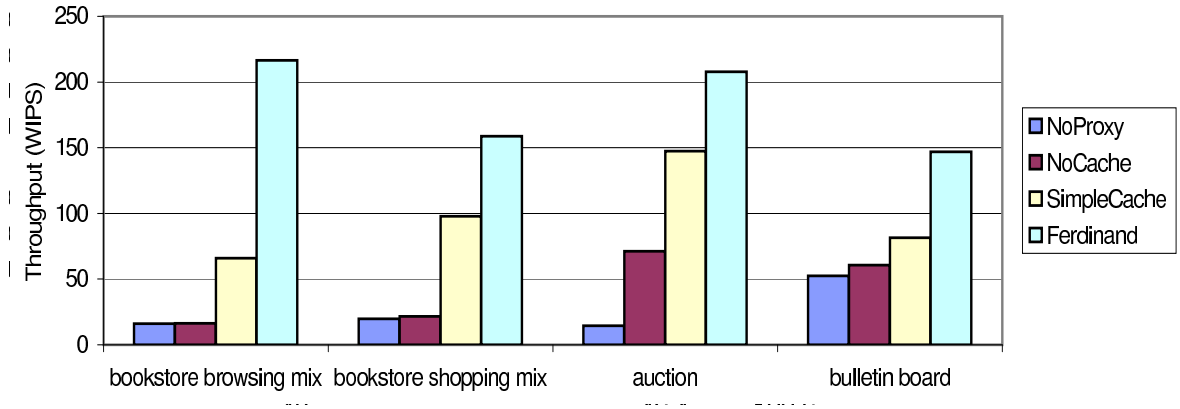


Figure 5.5: Throughput of Ferdinand compared to other scalability approaches.

5.3 The performance of cooperative query result caching

To evaluate the usefulness of cooperative caching we compared the performance of four approaches: (1) NOPROXY, a centralized three-tier system with no proxy servers and only the centralized database server cache, (2) NOCACHE, the CDN-like architecture that does not attempt to scale the home database server, (3) SIMPLECACHE, and (4) Ferdinand. As described in Section 4.4, we executed all of our scalability experiments on the Emulab network testbed [98]. For all of our workloads we partitioned user requests among the DBSS nodes, so that requests from each particular end user would always be executed at the same DBSS node as would occur from many typical load-balancing techniques. All scalability experiments used warm caches and DBSS sizes as described above in Sections 5.2.1 and 5.2.2.

Figure 5.5 shows the maximum throughput, under the response-time threshold, for each caching approach and each of our benchmarks, and Table 5.1 shows the average cache miss rates for SIMPLECACHE and Ferdinand, with the local cache miss rates and overall cache miss rates shown separately for Ferdinand. These miss rates are determined as the mean miss rate for each workload and system at near-peak throughput for three to five experimental executions, and have an uncertainty of about 0.5% each.

The performance of NOCACHE was only marginally better than NOPROXY for the data-intensive bookstore and bulletin board benchmarks. Replicating the web and application server significantly scaled the more computational auction benchmark, but the central database eventually became a per-

	SIMPLECACHE	Ferdinand local	Ferdinand overall
bookstore browsing mix	17%	14%	7%
bookstore shopping mix	22%	21%	14%
auction	40%	35%	17%
bulletin board	20%	18%	11%

Table 5.1: Cache miss rates for Ferdinand and SIMPLECACHE.

formance bottleneck even for this workload. SIMPLECACHE's scalability was much better, attaining better than four times the throughput than NOPROXY for both bookstore workloads. SIMPLECACHE's relative performance gain was not as significant for the auction or bulletin board benchmarks, but still significantly improved upon the throughput of the non-caching system. These results emphasize the importance of shielding queries from the database server to scale the overall system. For the auction benchmark, the cache miss rate is simply too high to support significant scaling, as Table 5.1 shows. For the bulletin board, SIMPLECACHE successfully shielded the database server from most database queries. Some bulletin board Web interactions, however, resulted in a large number of cache misses and required many rounds of communication between the proxy server and the central database. For those interactions, the many rounds of communication prevented a significant improvement of end-user latency, limiting the overall scalability although reducing load on the central database system.

Ferdinand outperformed SIMPLECACHE for all workloads. It achieved an overall factor of 13.4 scale-up on the bookstore browsing mix compared to the NOPROXY system – about a factor of 3 compared to SIMPLECACHE. For the auction benchmark Ferdinand improved throughput by 40% compared to SIMPLECACHE's traditional caching approach, gaining about a factor of 3 compared to the non-caching system. For the bulletin board Ferdinand improved throughput by about 80% compared to SIMPLECACHE, and a factor of about 2.5 compared to NOCACHE. Ferdinand's overall cache miss rate was substantially better than SIMPLECACHE for all workloads, indicating that even though a query result may be missing from a local cache it is likely that another proxy server is already caching the relevant result.

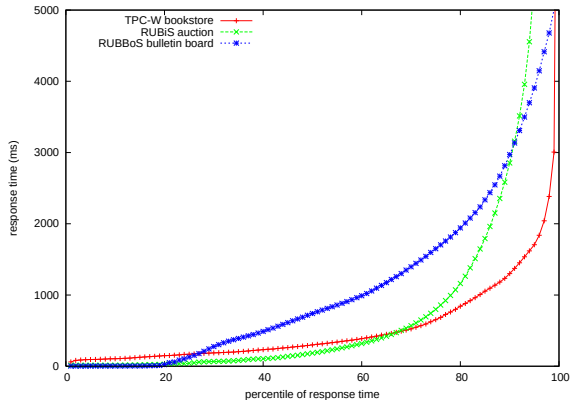


Figure 5.6: Cumulative distribution functions of Ferdinand's response time on our three benchmark applications.

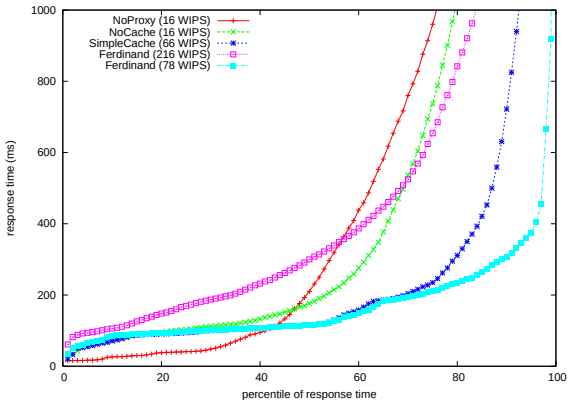


Figure 5.7: Cumulative distribution functions of our competing systems' response times on the TPC-W bookstore browsing mix.

Figures 5.6 and 5.7 show the cumulative distribution functions (CDFs) of the response time for various system configurations and benchmarks. In these graphs, the horizontal axis is the percentile of response time X, and the vertical axis denotes the response time R such that X% of Web requests were answered in R milliseconds. The goal of these figures is to give a more complete characterization of Ferdinand's response time than given by our throughput metric and its response time threshold.

Figure 5.6 shows the CDF for the response times of Ferdinand on each of our three benchmark applications at maximal throughput, using just the browsing mix for the bookstore benchmark. Even at maximal throughput (meeting the 90% response time guarantee of our throughput metric), about 80% of requests have latency under one second for the bookstore and auction benchmarks – with approximately 70% having latency under one-half second. A smaller proportion of requests have excellent latency for the bulletin board benchmark, but still nearly 60% of requests have latency under one second.

Figure 5.7 shows the CDF for the response times of our four competing systems on the TPC-W bookstore benchmark browsing mix at their maximal loads, as well as the CDF for Ferdinand at a "low" load that slightly exceeds the maximal load of any competing system. Be warned that the vertical axis of this figure is not scaled the same as Figure 5.6. NoProxy's performance at its maximal load results in very low latency for about half its requests, but relatively poor latency for the remainder. The

typical latency for the other systems is higher at their maximal loads, but the highest-latency requests are faster than with NOPROXY. SIMPLECACHE's typical latency outperforms that of NOCACHE, even though SIMPLECACHE is providing approximately four times the throughput in this experiment. At its maximal load, Ferdinand requires more time to reply to typical requests than SIMPLECACHE. However, it is important to remember that Ferdinand's maximal throughput is substantially higher than SIMPLECACHE's. When Ferdinand is handling a load comparable to SIMPLECACHE's maximum throughput, its typical response times are very competitive with either NOCACHE or SIMPLECACHE, and the slowest responses with Ferdinand at low load are much faster than those of either competing system. At this system load, Ferdinand responds to the vast majority of requests in under one-half second.

Ferdinand's high overall scalability is a tribute to the cooperative cache's ability to achieve high cache hit rates and Ferdinand's efficient lightweight caching implementation. With Ferdinand's minimal query processing, the cost of responding to a query at a Ferdinand proxy server is less than the cost of responding to a query in the NOPROXY system, even with MySQL's centralized query result cache. Overall, our results show that Ferdinand succeeds in scaling system throughput without hindering performance for typical requests.

5.3.1 Effect of distributed caching on local cache miss rates

In Section 3.3.2 we briefly discussed an accounting anomaly due to our use of shared storage for the local and distributed caches at each Ferdinand node. Because of this shared storage, a local query Q at its master Ferdinand node might result in a cache hit even if the node has not locally executed Q since its last invalidation. This is because Q might have been executed at a remote server and placed in the distributed cache at its master node, thus shared with master node's local cache. Note that the local cache miss rate in Ferdinand is less than the cache miss rate in SIMPLECACHE for all workloads, as we would expect from our use of shared storage.

In this section, our goal is to evaluate whether the discrepancy between the Ferdinand local cache miss rate and the SIMPLECACHE miss rate can be attributed to our shared storage, or whether it could possibly indicate a systemic flaw in our experimental methods. For example, the discrepancy could indicate that

	discrepancy expected	discrepancy measured
bookstore browsing mix	1.4%	2.2%
bookstore shopping mix	2.4%	1.8%
auction	4.4%	5.0%
bulletin board	2.2%	2.2%

Table 5.2: Computed and measured discrepancy between Ferdinand local cache and SIMPLECACHE miss rates.

the SIMPLECACHE caches were inadequately warmed compared to the Ferdinand caches. To do so, we create a rough analytical model of the local and distributed caches to compute the magnitude of the expected discrepancy for each workload, to confirm that the measured discrepancies are as expected and not indicative of an underlying problem with our evaluation.

Consider the discrepancy for a query Q at a Ferdinand node. The accounting anomaly occurs when the following events have happened: ($E1$) a local cache miss would have occurred had separate storage been used, ($E2$) the local node is the master node for Q, and ($E3$) another node has executed Q since it was last invalidated, causing Q to be in the distributed cache.

The second event should be independent of the others, because Q's master node is determined by its hash in the DHT and is not related to any workload factors. This means that the probability of the accounting anomaly is just $\Pr[E2] \cdot \Pr[E1] \cdot \Pr[E3|E1]$. Furthermore, because the DHT should assign queries to nodes in a way that balances the load in the distributed cache, we have the expected value of $\Pr[E2] = \frac{1}{n}$, taken over all queries in a DBSS of n nodes. Similarly taken over all queries, we have the expected value of $\Pr[E1]$ to be just the miss rate for queries in the SIMPLECACHE DBSS.

To compute the expected probability of the anomaly, therefore, we just need to compute that of $\Pr[E3|E1]$, the expected probability that Q was executed at another DBSS node before the local cache miss would have occurred, given that the local cache miss did occur. This is just the probability that Q is being executed at its master node first since its last invalidation. If we assume that the node at which a query is executed is independent of where it has been executed previously, then this event has expected

probability of $\frac{1}{n}$, yielding $\Pr[E3|E1] = \frac{n-1}{n}$. For many queries and workloads we expect this assumption to be true, that each query will be relatively uniformly distributed among the DBSS nodes. The assumption is clearly not universally true, however, and is even false for some queries in our workloads since we assign each user to the same DBSS node each time they submit a new Web request. However, because this computation is an expected value across all queries in a workload, we believe that the result is a reasonable approximation of the actual probability.

Overall, this yields an approximate calculation for the anomaly occurring for each submitted query with probability $\frac{n-1}{n^2} \cdot m$, where m is the SIMPLECACHE miss rate for the workload and n is the number of DBSS nodes.

We use this approximation to compute the expected discrepancy for each our experimental workloads. Table 5.2 shows the discrepancy we expect from our analytical model, as well as the discrepancy we measured between the Ferdinand and SIMPLECACHE experiments. In all cases, the expected discrepancy is very similar to the actual difference measured, with the measured value being never greater than that expected within the uncertainty of the cache miss rate measurements.

This analysis supports that the difference between the local cache miss rate in Ferdinand and the overall cache miss rate in SIMPLECACHE is due to our shared disk-based cache, and not due to a systemic experimental error.

5.4 Ferdinand in higher-latency environments

Our next experiment models a CDN-like deployment of Ferdinand in which the proxy servers are placed in various positions on the Internet and are distant from each other and the central database. We use the Emulab testbed to emulate high latency connections among the DBSS nodes and the home database server. The maximum throughput of both Ferdinand and SIMPLECACHE changes in such an environment, as the response times get too long and exceed the latency thresholds. Ferdinand's performance, however, might be additionally hindered since a query that misses at both the local and master caches requires multiple high-latency network hops rather than just one.

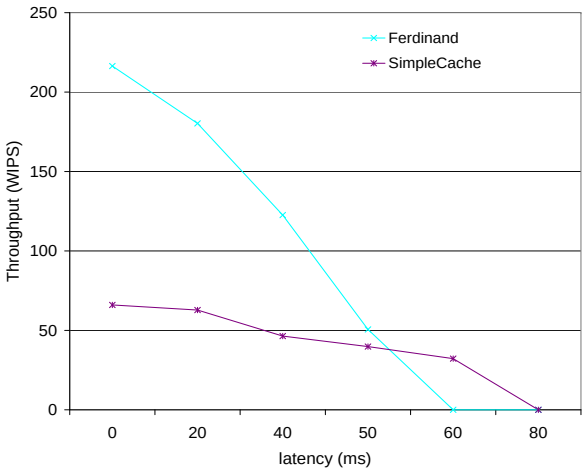


Figure 5.8: Throughputs of Ferdinand and SIMPLECACHE as a function of round trip server-to-server latency.

Figure 5.8 compares the throughput of SIMPLECACHE to Ferdinand as a function of server-to-server round trip latency, for the bookstore browsing mix. As network latency increased the performance of both caching systems decreased, as expected. The magnitude of performance degradation was worse for Ferdinand than for SIMPLECACHE as feared. Even though Ferdinand suffers fewer cache misses than SIMPLECACHE, the cost of these cache misses becomes prohibitive as the network latency increases. For low- and medium-latency environments, however, Ferdinand continued to significantly outperform the traditional caching system.

The performance of both Ferdinand and SIMPLECACHE in high latency environments is largely a consequence of the standard throughput metric that we used. Most servlets in the bookstore benchmark execute just one or two database requests, but several servlets (the bookstore "shopping cart," for instance) sometimes execute many. As the latency between the application server and database server increased it was increasingly difficult for the servlet to successfully respond within the threshold for that interaction type, and the overall system needed to be increasingly lightly loaded to succeed. In this environment the response times of most interactions were still far below their latency thresholds. With an 80 millisecond server-to-server round trip latency, however, neither Ferdinand nor SIMPLECACHE could routinely respond to the shopping cart servlet successfully within its 3-second time bound. With 60 millisecond server-to-server latency SIMPLECACHE succeeded by a small margin at low system loads,

while Ferdinand always required at least 3.2 seconds each for the slowest 10% of shopping cart interactions and thus did not qualify by the standards of the throughput metric. Even in this environment, however, Ferdinand could sustain high loads (over 100 WIPS) while only slightly violating the latency bounds. If an application designer was aware of Ferdinand, he might rewrite these applications to reduce the number of queries needed to produce the shopping cart page or even execute independent queries in parallel.

Overall, Ferdinand achieved much greater scalability than all other database query-caching methods of which we are aware. The performance of our SIMPLECACHE implementation is similar to other work in this area, and it achieved cache hit rates and throughput gains comparable to previous query-caching systems [62]. In low latency network environments Ferdinand outperformed SIMPLECACHE on all workloads we examined by providing a higher throughput for the same response-time bound. Our results also demonstrate that DHT-based cooperative caching is a feasible design even in some medium-latency environments expected for a CDN-like deployment. Finally, note that Ferdinand improves the price-to-performance ratio compared to SIMPLECACHE for most environments since both systems use the same hardware configuration.

5.5 Conclusions

The central database server is often the performance bottleneck in a web database system. Database query caching is one common approach to scale the database component, but it faces two key challenges: (1) maintaining a high cache hit rate to reduce the load on the central database server, while (2) efficiently maintaining cache consistency as the database is updated.

This chapter evaluated Ferdinand, the first proxy-based cooperative database query cache with fully distributed consistency management for web database systems. To maintain a high cache hit rate and shield queries from the central database server, Ferdinand uses both a local database query cache on each proxy server and a distributed cache implemented using a distributed hash table. Each proxy's cache is maintained as a simple disk-based map between each database query and a materialized view of that

query's result. To efficiently maintain cache consistency Ferdinand uses a scalable topic-based publish / subscribe system.

To evaluate Ferdinand we implemented a fully functioning prototype of both Ferdinand and common competing approaches to database query caching and consistency management. We then used several standard web application benchmarks to show that Ferdinand attains significantly higher cache hit rates and greater scalability than the less sophisticated designs.

Chapter 6

Publish / Subscribe for the Consistency Management of Web Database Caches

In Ferdinand, when a DBSS node caches a database query result, it subscribes to publish / subscribe messages related to the cached query. When a node updates the database, it publishes the update to the publish / subscribe system, which then notifies any DBSS nodes that are possibly affected by the update. The publish / subscribe system therefore encapsulates the process of matching updates to affected queries and delivering update notifications to the DBSS nodes that cache those queries.

As mentioned in Section 1.4.2, one major advantage of using publish / subscribe for consistency management is that it allows us to leverage existing technology for communicating updates to the Ferdinand caches. However, it introduces several implementation challenges. First, there are a wide variety of publish / subscribe systems, supporting different degrees of scalability and different methods of matching publications to subscriptions. The process of matching database updates to the queries they affect is more complex than the matching process supported by most existing publish / subscribe systems, and it is not clear what type of system we should use. To compound the problem, the systems that support more complex matching operations are less scalable, and there is a fundamental trade-off between the complexity of the publish / subscribe matching process and the scalability of the implementation [96]. Second, once we have selected a particular publish / subscribe system, it is not clear what publications and subscriptions should be made to ensure that a DBSS node is notified of all relevant updates.

In this chapter we show that even the simplest publish / subscribe paradigm – topic-based publish / subscribe – is sufficient to implement Ferdinand's consistency management system. The key idea to our approach is that we use foreknowledge of the Web application and its associated database requests to constrain the range of queries and updates that the Web application might execute. By analyzing the Web application offline, we can efficiently match updates to the queries they affect using even simple topic-based publish / subscribe, even though the relationship between updates and affected queries can be complex.

We start in Section 6.1 by describing the state-of-the-art in various publish / subscribe paradigms, discussing the various design factors and arguing that our use of topic-based publish / subscribe is a reasonable choice.

We continue in Section 6.2 by introducing the Query / Update Multicast Association (QUMA) problem, the problem of determining to what topics a Ferdinand node subscribes when a query is cached and to what topics Ferdinand publishes each update. There we detail how we use foreknowledge of the Web application to solve the QUMA problem, introduce a variant of our first QUMA solution, describe an experimental framework to compare the two approaches, and evaluate them.

Section 6.3 discusses our implementation of consistency management within Ferdinand. There we prove that our use of publish / subscribe ensures that an Ferdinand cache will receive all update notifications that affect it, and show that our publish / subscribe-based consistency management far outperforms the simpler broadcast-based methods of alternative scalability systems.

6.1 The paradigms of modern publish / subscribe

In general, a publish / subscribe system is any system that supports two basic operations: a *subscribe* operation in which a client subscribes to a set of messages, and a *publish* operation in which a client publishes a message. The publish / subscribe system encapsulates the work of matching publications to subscriptions and delivering the publications to subscribed clients.

Today there are two major paradigms of publish / subscribe: topic-based publish / subscribe and

content-based publish / subscribe. In topic-based publish / subscribe, each subscription specifies a fixed topic, and a subscribed client receives all publications that specify the same fixed topic. In content-based systems, subscriptions can place constraints on the message contents in addition to specifying a topic name. Consider a stock-ticker application in which a client periodically publishes the current price of a stock on a topic that matches the stock name, e.g. Google's stock GOOG. Using topic-based publish / subscribe, a client would be constrained to subscribing only to the topic name, GOOG, and then would receive all publications of Google's current stock price. Using content-based publish / subscribe, however, a client could choose to only receive publications that match a specified criterion. For example, a client might only be interested in the case when Google's stock price exceeds $500, in which they might enter a subscription of the form *subscribe*(PRICE $>$ 500).

In [96], Uhl showed that there is a fundamental trade-off between the scalability and expressibility of publish / subscribe systems. This means that the most scalable content-based publish / subscribe system can be no more scalable than the most scalable topic-based publish / subscribe system.

Currently, Scribe [84] is one of the most scalable topic-based publish / subscribe systems. In Scribe, each topic has a root node, as determined by the hash of the topic name in the Pastry DHT. Subscribers to a topic are organized into a subscription tree rooted at the topic's root node, with new subscribers added greedily to the first Pastry node in the subscription tree they encounter when their subscription request is routed toward the root. This design allows Scribe to support a high subscription and unsubscription rate by diffusing subscription hot spots, spreading the hot spot among nodes near the hot topic's root.

Scribe's design is mimicked by one of the most scalable content-based publish / subscribe systems, Hermes [78]. Hermes also assigns each topic a root node in a DHT. In Hermes, however, all subscriptions and unsubscriptions to a topic are forwarded all the way to the topic's root, which organizes subscribers into a hierarchical structure to perform subscription-publication matching based on the contents of published messages. This difference enables a more complex matching process than supported by Scribe, but creates subscription and unsubscription hot spots.

In our next section we will describe how we analyze the Web application to match updates to the queries they potentially affect. In most cases, a simple topic-based publish / subscribe system can match

the related subscriptions and publications just as well as a content-based publish / subscribe system could. The most notable exception is when a query or update uses a selection predicate based on the inequality of relational values. In this case, a content-based publish / subscribe system might allow us to exactly match an update to the queries it affects, whereas a topic-based publish / subscribe system might require us to broadcast the update to all Ferdinand nodes.

Because Ferdinand nodes frequently cache and invalidate query results, however, they subscribe to and unsubscribe from various notification messages at a high rate. Although content-based publish / subscribe would allow us to better match updates to affected queries in same cases, we expect that advantage to be rare, whereas the advantage of topic-based systems would be realized on nearly every subscription and unsubscription request. Because of this difference, we chose to implement Ferdinand's consistency management using just topic-based publish / subscribe, rather than implement and empirically evaluate consistency management using both publish / subscribe paradigms.

6.2 The query / update multicast association (QUMA) problem

The key challenge in efficiently maintaining cache consistency is to minimize the amount of inter-proxy communication while ensuring correct operation. Ideally, any update notifications published to a topic should affect each database query subscribed to that topic. Good database query / update multicast associations also avoid creating subscriptions to topics to which updates will never be published and publications to topics for which no queries have caused subscriptions. An additional goal is to cluster related queries into the same topic, so that an update that affects two or more queries requires only a single notification.

As a practical matter it is inefficient to have a topic for every underlying object in the database since database requests often read and update clusters of related data as a single unit. In such a case it is better to associate a single topic with the whole data cluster so that reads or updates to the cluster require only a single subscription or publication. Thus, the key to obtaining a good QUMA solution is to accurately cluster data for a given database workload. In the next section we describe how to efficiently achieve

Template U1: `INSERT INTO inv VALUES (id = ?, name = ?, qty = ?, entry_date = NOW())`

Template U2: `UPDATE inv SET qty = ? WHERE id = ?`

Template Q3: `SELECT qty FROM inv WHERE name = ?`

Template Q4: `SELECT name FROM inv WHERE entry_date > ?`

Template Q5: `SELECT * FROM inv WHERE qty < ?`

Figure 6.1: Database requests for an example inventory application.

this goal for a given web application using offline analysis of its database requests.

6.2.1 Using offline analysis to solve the QUMA problem

The database requests in many web applications consist of a small number of static templates within the application code. Typically, each template has a few parameters that are bound at run-time. These templates and their instantiated parameters define the data clusters that are read and updated as a unit during the application's execution. To enable consistency management for a given web application we first inspect that application's templated database requests. For each database query-update template pair we then extend techniques from offline database query-update independence analysis [37] to determine for which cases the pair are provably independent, i.e. for which cases instantiations of the update template do not affect any data read by instantiations of the database query template.

To illustrate this analysis consider the database requests for an example inventory application, shown in Figure 6.1. The example application consists of two update templates and three database query templates, each with several parameters. Template U1 affects instantiations of Template Q3 when the same name parameter is used. Template U1 also affects any instantiation of Template Q4 for which the entry date was in the past, and instantiations of Template Q5 whose quantity parameter was greater than that of the newly inserted item. Template U2 affects instantiations of Template Q3 whose name matches the id parameter that was used, is independent of Template Q4, and affects instantiations of Template Q5 if

Template	Associated Topics
Template U1	{TOPICU1:NAME=?, TOPICU1}
Template U2	{TOPICU2}
Template Q3	{TOPICU1:NAME=?, TOPICU2}
Template Q4	{TOPICU1}
Template Q5	{TOPICU1, TOPICU2}

Figure 6.2: A correct QUMA solution for our sample inventory application.

the change in the item's quantity traverses the database query's quantity parameter.

Consider a topic based on the data affected by instantiations of Template U1, called TOPICU1. Correct notification will occur for updates from this template if queries of Template Q3, Template Q4, and Template Q5 all create a subscription to TOPICU1 and all such updates publish to TOPICU1. A slightly better solution, however, is to additionally consider parameter bindings instantiated at runtime and use a more extensive data analysis, noting that an update of Template U1 affects a database query of Template Q3 only if they match on the "name" parameter. To take advantage of this fact we can bind the value of the "name" parameter into the topic at runtime. In this association, queries of Template Q1 subscribe to TOPICU1:NAME=? for the appropriate parameter binding. Updates published to the bound topic TOPICU1:NAME=? will then result in a notification only if the name parameters match, reducing the number of unnecessary notifications. The disadvantage of this approach is that the number of possible topics is proportional to the template parameter instances instead of to the number of database query-update pairs. Fortunately, existing publish / subscribe systems efficiently support large numbers of topics.

Figure 6.2 shows a correct QUMA solution for our sample inventory application. A question mark in the topic name indicates that the appropriate parameter should be bound at runtime when the template is instantiated. Suppose that proxy server *A* starts with a cold cache. If proxy *A* caches a database query of Template Q3 with the parameter "fork" it would then subscribe to the topics TOPICU1:NAME=FORK and TOPICU2. If proxy server *B* then used Template U1 to insert a new item with the name "spoon"

Template	Topic-by-update	Topic-by-query
Template U1	{TOPICU1:NAME=?, TOPICU1}	{TOPICQ3:NAME=?, TOPICQ4, TOPICQ5}
Template U2	{TOPICU2}	{TOPICQ3, TOPICQ5}
Template Q3	{TOPICU1:NAME=?, TOPICU2}	{TOPICQ3, TOPICQ3:NAME=?}
Template Q4	{TOPICU1}	{TOPICQ4}
Template Q5	{TOPICU1, TOPICU2}	{TOPICQ5}

Figure 6.3: Topic-by-update and Topic-by-query QUMA solutions for our sample inventory application.

then *B* would publish update notifications to the topics TOPICU1:NAME=SPOON and TOPICU1.

In practice, the solution we describe can efficiently map queries and updates to topics when those queries and updates use only equality-based selection predicates, because the parameters embedded at run time often yield a precise match between the updates published to such topics and the queries affected. Database requests using range or other selection predicate types, joins, or aggregates are typically mapped to topics that do not embed run time parameters, resulting in a higher number of unnecessary update notifications. A content-based publish / subscribe system would additionally be able to match queries and updates that use range, inequality, or other selection predicates, but would be similarly inefficient for joins and aggregates as topic-based publish / subscribe.

6.2.2 Another QUMA solution using offline analysis

In the example QUMA solution of Figure 6.2, we had a topic for each update template, sometimes with additional update-related topics to embed run-time parameters. In that solution, a query would cause a subscription to an update's topic if the query was dependent on data affected by that update.

A similar strategy is to instead have a topic for each query template, with additional query-related topics to embed run-time parameters. In this solution, a query causes a subscription to its own topics. An update publishes a notification to the topic for every query that depends on data affected by the update. This new strategy relies on the same offline analysis described in Section 6.2.1. We call the first

strategy – with a topic for each update template – *Topic-by-update*, and call this new strategy *Topic-by-query*.

The practical difference between Topic-by-update and Topic-by-query is subtle. Figure 6.3 gives the QUMA solutions for both strategies for our example inventory application (Figure 6.1). If all queries were affected by exactly one update and all updates affected exactly one query, then Topic-by-query and Topic-by-update would yield equivalent QUMA solutions, except for a difference in the topic names. In practice, however, queries are affected by multiple updates and updates can affect multiple queries. Because of this fact, the different strategies might need different numbers of subscriptions for each query and different numbers of publications for each update. A good example of this difference is Template Q5 in our example application. For Topic-by-update a query of this template must subscribe to two topics, one for each update that affects it. For Topic-by-query, however, it need subscribe only to one topic: its own.

On average, we expect (1) queries to depend on fewer topics with Topic-by-query than with Topic-by-update since multiple update templates may affect a query template. Similarly, we expect (2) updates to typically publish to more topics with Topic-by-query than with Topic-by-update. Finally, we expect (3) that Topic-by-update is more likely to aggregate related queries into the same topic since queries dependent on the same underlying data would tend to be affected by the same update templates.

Because of (2), we expect Topic-by-update to result in a lower publication rate than Topic-by-query for most workloads. However, most workloads are dominated by queries rather than updates, and the overall subscription rate is dependent on the competing factors (1) and (3). For workloads where Topic-by-update's aggregation of related queries is low, we expect Topic-by-query to result in a substantially better subscription rate. The relative performance of the two configurations is unclear when Topic-by-update succeeds in aggregating related queries into the same topic.

6.2.3 Using simulation to evaluate QUMA solutions

To evaluate our QUMA solutions in a wide range of system sizes, we built a simulator that allows us to test very large network configurations. Our simulator is a Java program that models the cache state and

network activity of the SIMPLECACHE DBSS (introduced in Section 5.1.1), taking as input a collection of traces of database requests and modeling the SIMPLECACHE behavior as if it executed those requests.

Each input trace is a log of database requests and the time at which each request was issued. For each trace, the simulator models SIMPLECACHE's behavior as if that trace were executed at a single proxy node. The cache state at each proxy and the membership of every publish / subscribe topic are exactly traced through a simulation execution. The simulator processes each database request atomically and interleaves the traces based on the requests' time stamps. No actual database activity occurs and the simulator does not impose any additional constraints on the ordering of simulated database requests. The simulator models an unbounded cache, because our replacement algorithm is not executed in practice during our regular experiments. Each input trace was created by logging 5-10 minutes of real benchmark database activity, without using a DBSS or database cache.

Although it would have been more accurate to simulate the Ferdinand DBSS rather than SIMPLECACHE, simulating SIMPLECACHE is much less complex because our simulation does not need to include the placement of items into Ferdinand's distributed cooperative cache. Most of Ferdinand's consistency traffic is used for the local caches, not the cooperative cache, so the discrepancy between the two systems should be slight. We expect that our simulations using SIMPLECACHE accurately characterize Ferdinand's publish / subscribe performance.

A final key difference between our simulation results and our actual Ferdinand experiments is that our TPC-W benchmark in the simulation framework uses a significantly smaller database size, with only 10,000 items for sale and 217 MB of data, compared to our earlier experiments (in Sections 5.3 and 5.4) of 1 million items and 480 MB of data.

In our simulations, each simulated DBSS node supports the activity of 160 simultaneous emulated browsers. Thus, in every experiment the overall system load is proportional to the number of DBSS nodes. As our primary interest is the steady-state behavior, each DBSS node starts with a warm cache derived from the execution of other input traces of the appropriate benchmark. For reasons of practicality, each proxy cache was warmed using multiple database traces of short length rather than a single trace of long length. Although the total length of these traces approximates the length of a single trace of

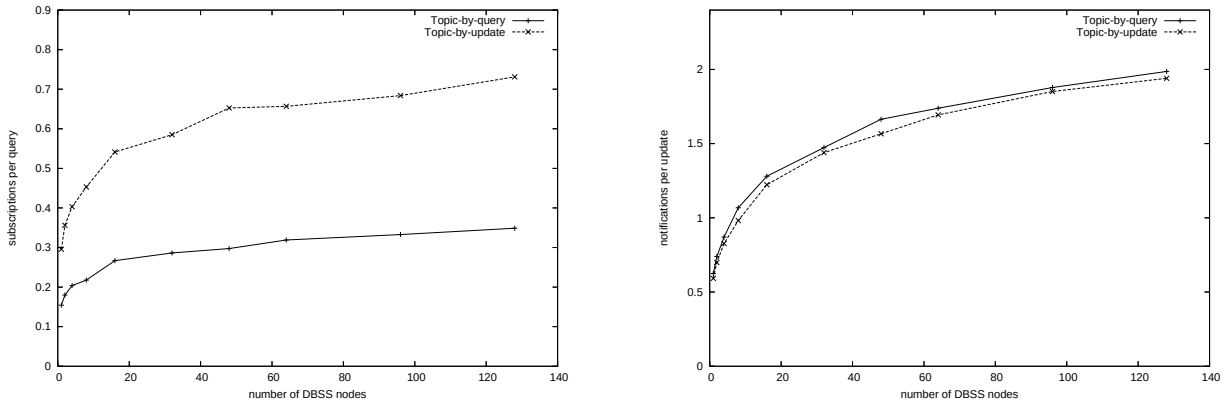


Figure 6.4: Simulated performance of our QUMA solutions on the TPC-W bookstore benchmark.

several-hour duration, the short traces may bias the cache states toward queries that are executed early in the benchmark workloads, slightly biasing the cache performance in the simulation results.

6.2.4 Comparing our Topic-by-update and Topic-by-query QUMA solutions

Figure 6.4 shows the number of publish / subscribe subscription messages and notifications delivered for the TPC-W bookstore benchmark, for SIMPLECACHE DBSS configurations with 1 to 128 nodes. This data is normalized by the number of requests, presented as the number of subscriptions per query and the number of update notifications delivered per update. Recall that our goal is to minimize the number of network messages, so smaller numbers here are better.

For TPC-W, Topic-by-update requires twice as many subscriptions per query as Topic-by-query, regardless of the network size. This fact is not too surprising since each query in TPC-W depends on nearly twice as many topics when using Topic-by-update. This indicates that Topic-by-update does not successfully aggregate related queries for TPC-W. Topic-by-update and Topic-by-query require similar number of update notifications for the TPC-W bookstore, so overall, Topic-by-query requires fewer publish / subscribe messages for this workload. Also notable is the small number of notifications required per update for TPC-W, even for very large network sizes. This fact shows that for the bookstore benchmark, our publish / subscribe-based consistency management is highly efficient, requiring a tiny fraction of the number of messages that would be required by more traditional broadcast-based methods. This

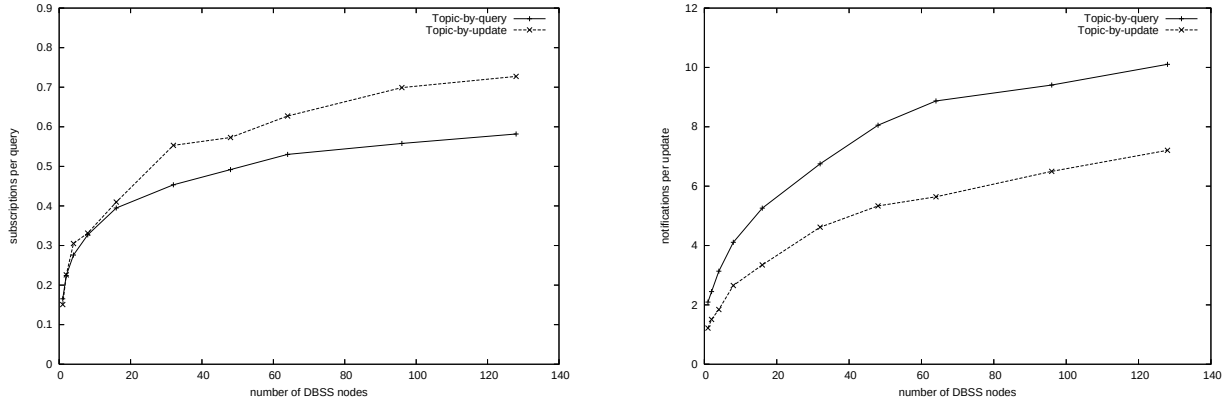


Figure 6.5: Simulated performance of our QUMA solutions on the RUBiS auction benchmark.

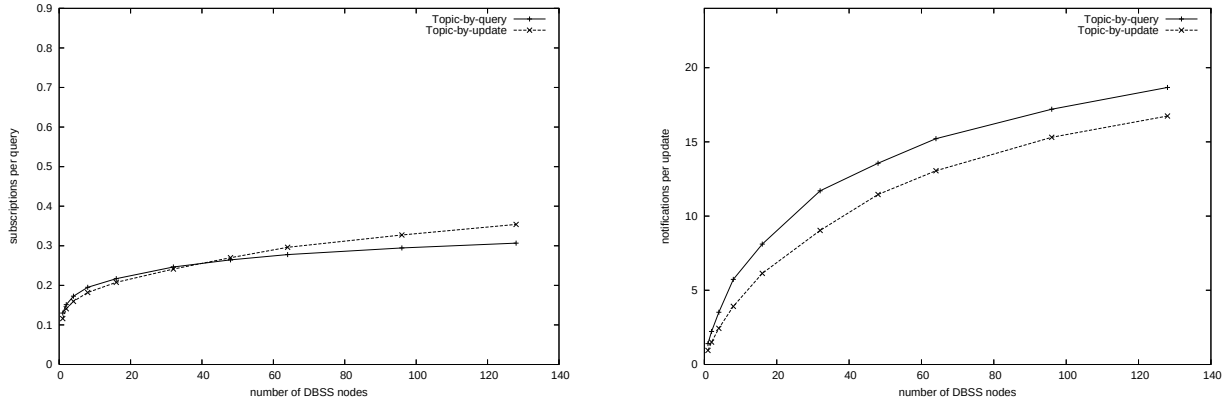


Figure 6.6: Simulated performance of our QUMA solutions on the RUBBoS bulletin board benchmark.

high efficiency is a result of the fact that most queries and updates in TPC-W are simple and typically retrieve or update a single row, having chosen that row using an equality-based selection predicate.

Figures 6.5 and 6.6 show the results of similar experiments using the RUBiS auction and RUBBoS bulletin board benchmarks, respectively. For RUBiS, Topic-by-query again outperforms Topic-by-update for the subscription metric, requiring nearly 40% fewer subscriptions per query for large DBSS sizes. Topic-by-update, however, requires substantially fewer update notifications, improving upon Topic-by-query by almost 35% for some configations. Although both Topic-by-query and Topic-by-update substantially outperform broadcast-based invalidation (by as much as a factor of 12 and 16, respectively), our consistency management system requires more messages per update for RUBiS than for TPC-W; this

is because in RUBiS, updates are far more likely to affect global data than in TPC-W. Overall, though, the auction benchmark requires many fewer updates (compared to the number of queries) than TPC-W.

For the RUBBoS bulletin board benchmark, both Topic-by-query and Topic-by-update perform equally well for subscriptions, with Topic-by-update even outperforming Topic-by-query in some cases for small DBSS networks. This shows that Topic-by-update does successfully aggregate related queries for this workload. Notably, this effect is reduced for large networks, presumably because the higher relative update rate reduces the probability of any related queries already being cached. Compared to broadcast-based invalidation, our consistency management mechanism performs the worst for RUBBoS, but publish / subscribe-based system still requires only about 15% of the update notifications that would otherwise be broadcast to the DBSS nodes. Here, again Topic-by-update slightly outperforms Topic-by-query by the update notification metric, but again, the RUBBoS bulletin board issues far more database queries than updates.

Overall, our simulator demonstrates that neither Topic-by-query nor Topic-by-update dominates the other for all workloads. For the TPC-W bookstore, Topic-by-query is better since it significantly outperforms Topic-by-update for database queries but is only slightly worse for database updates, and the overall workload is query-dominated. For the RUBiS auction and RUBBoS bulletin board the analysis is not so clear. For these benchmarks Topic-by-update might perform better for some network sizes and numbers of users, but worse as the overall system load increases.

These experimental results demonstrate that for peak efficiency, the DBSS would need to evaluate consistency management performance for both QUMA strategies on an application-by-application basis. In some cases an administrator might be able to statically evaluate whether Topic-by-query or Topic-by-update is preferable for a given workload. The key static property that affects the performance of each strategy is the typical number of queries affected by each update and the typical number of updates that affect each query in the application. For example, if a query is affected by many updates but each update affects few queries, then Topic-by-query might be preferable: each query would result in only subscriptions to its own topics, and each update would require few publications. In contrast if Topic-by-update were used for this application, then each query might require a subscription to many topics – one

for each update that affects it – without any reduction in the number of publications required for each update.

6.3 Consistency management for Ferdinand

Like most other modern database query-caching systems, Ferdinand relaxes consistency for multi-statement transactions and therefore is ill-suited for circumstances where multi-statement transactional consistency is strictly required. Ferdinand simply ensures that the contents of each database query cache remain coherent with the central database server, guaranteeing full consistency when all transactions execute only a single database request. Ferdinand's consistency management system uses the offline workload analysis described above, and our implementation uses the Topic-by-update QUMA solution described in Sections 6.2.1 and 6.2.2. We manually generated the QUMA solution for each of our benchmark applications, but we believe the process is easily automated and are developing an application to do so.

In our implementation, for each topic there is a second "master" topic used for communication with master proxies. When a local cache miss occurs at a proxy server the proxy subscribes to all non-master topics related to the database query and waits for confirmation of its subscriptions before forwarding the database query to its master proxy server. A master proxy server for a database query similarly subscribes to all related master topics and waits for confirmation before forwarding the database query to the central database.

When an update occurs, notifications are first published to only the master topics for the update. Any master proxy servers for affected queries will receive the update notification, invalidate the affected queries and re-publish the update notification to the corresponding non-master topics, to which any other affected proxy servers will be subscribed.

This hierarchical implementation with master and non-master topics is necessary to correctly maintain consistency because publish / subscribe does not constrain the order of notification delivery to different subscribers to the same topic. If we did not use these master topics, it would be possible for a non-

master proxy to invalidate a database query's result and re-retrieve a stale copy from the master before the master receives the notification. If this were to occur, the non-master could cache the stale result indefinitely.

This section continues by proving that our consistency management system prevents stale database query results from being indefinitely cached at any proxy server, in Section 6.3.1. In Section 6.3.2 we evaluate the performance contribution of publish / subscribe-based consistency management in Ferdinand by comparing our implementation to an approach that instead broadcasts updates to each DBSS node.

6.3.1 The correctness of Ferdinand's consistency management

We have proven that given a reliable underlying publish / subscribe system, our consistency management design prevents stale database query results from being indefinitely cached at any proxy server. Specifically the publish / subscribe system must have the following two properties: (A1) When a client subscribes to a publish / subscribe topic, the subscription is confirmed to the client, and (A2) A subscribed client is notified of all publications to a topic that occur between the time the client's subscription to the topic is confirmed and the time the client initiates an unsubscription from the topic. We also assume (A3) that the central database provides a slightly stronger guarantee than standard one-copy serializability, called *order-preserving serializability*: for transactions T_1 and T_2 at the central database such that T_1 commits before T_2 begins, the database produces a serial ordering of operations such that there is an equivalent serial ordering of transactions such that T_1 appears before T_2. These properties enable us to prove the following theorem:

Theorem 1. *Let q be the result of some query Q executed at time t_Q and let U be any later update affecting Q executed at time $t_U > t_Q$, with t_Q and t_U defined by the serial ordering of transactions at the central database server. Query result q will be removed from any Ferdinand proxy that is already caching or will ever receive q.*

We prove this theorem below, but first summarize the proof here. We use the above reliability guar-

antee to first show that all master proxies caching q will receive an update notification invalidating q, and then show that all other proxies caching q will receive a similar update notification. The publish / subscribe system's confirmation of subscriptions and reliability guarantee enables us to constrain event orderings at the proxy servers. When combined with Ferdinand's algorithms (described in Section 3.4) and serializability guarantees at the central database those properties enable us to prove that race conditions cannot prevent invalidation of stale cached query results. As the back-end database server progresses through series of states, Ferdinand guarantees that each materialized query result at each cache will also progress, possibly skipping some states. The view at a cache can be stale briefly but no view can remain stale indefinitely.

We prove Theorem 1 in two parts. We first show that a query q will be invalidated from its master proxy server, and then prove a similar guarantee for any other proxies that ever receive q.

Lemma 1. *Let q, Q, and U be as above. Then query result q will be invalidated at its master proxy by Ferdinand's consistency mechanism.*

Proof. Since U affects Q our query / update multicast association guarantees that there exists at least one master group G such that Q ensures subscription to G and U publishes a notification to G.

Consider the time t_{begin} at which the Q was begun at the central database, and the time t_{commit} at which U was committed. By assumption A3 we have $t_{begin} < t_{commit}$ since Q appears before U in the serial ordering of transactions at the central database.

Let t_{sub} be the time at which Q's master proxy has confirmed its subscription to G (Algorithm 3.2, step 08) and let t_{pub} be the time at which notification of U is published to G (Algorithm 3.3, step 05), as viewed by Q's master proxy. We then have that $t_{sub} < t_{begin} < t_{commit} < t_{pub}$ since subscription to G was confirmed before Q was submitted to the central database server and publication to G occurred after U's commit was confirmed by the central database.

Suppose that Q's master proxy has received no invalidations for q by time t_{pub}. Then the master proxy will still be subscribed to G at time t_{pub} since proxies never unsubscribe from groups on which cached or pending queries depend. By assumption A2 we have that the master proxy will receive notification of

U. □

We now show that query result q will be invalidated at all local non-master caches. The proof is highly similar to Lemma 1.

Lemma 2. *Let C be any non-master proxy for Q that is already caching or will ever receive query result q. Then C will receive an invalidation of q.*

Proof. Let U' be the update that invalidates q at Q's master proxy. (This need not be the same update U in Theorem 1.) Since U' invalidated q we have that U' affects Q and our query / update multicast association guarantees that there exists at least one regular non-master group G such that Q ensures subscription to G and Q's master proxy will republish the invalidation to G.

Let t_{sub} be the time at which C has confirmed its subscription to G (Algorithm 3.1, step 08) and let t_{pub} be the time at which Q's master publishes its invalidation to G (Algorithm 3.4, step 11). If q is valid at the master proxy we have that $t_{sub} < t_{pub}$ since the master proxy marks all cached and pending queries as invalid before republishing the notification to the non-master groups (Algorithm 3.4, step 11). The only circumstance in which a master proxy returns an already invalid result q to a local proxy is if the query Q was pending on behalf of that local proxy when the master received the invalidation. In this case we have $t_{sub} < t_{begin} < t_{commit} < t_{pub}$ as in Lemma 1, also yielding $t_{sub} < t_{pub}$.

Much as in Lemma 1, suppose that C has received no invalidations for q by time t_{pub}. By the same logic we have that C will receive a notification of U' and invalidate q. □

Lemma 1 and Lemma 2 trivially combine to prove Theorem 1.

6.3.2 The performance of publish / subscribe consistency management in Ferdinand

We determined the performance contribution of publish / subscribe as the mechanism for propagating update notifications within Ferdinand. Previous query-caching work has focused primarily on two alternative approaches: (1) BROADCAST-based designs in which the central database broadcasts each update

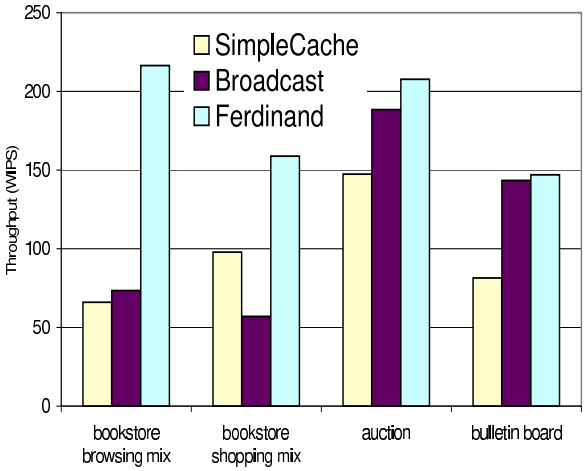


Figure 6.7: Throughput of Ferdinand compared to broadcast-based consistency management.

to every proxy server, and (2) a directory-based approach in which the central server tracks (or approximately tracks) the contents of each proxy's cache and forwards updates only to the affected proxy servers. The development of efficient directories for consistency management is part of our on-going work and here we compared Ferdinand just to the broadcast-based approach.

We implemented BROADCAST within a Ferdinand-like system. We remove our publish / subscribe-based consistency management and had the central database server instead broadcast each update notification to every proxy server. Like Ferdinand, BROADCAST uses DHT-based cooperative query result caching so that we are only comparing their invalidation mechanisms.

Our Ferdinand publish / subscribe-based consistency management uses our Topic-by-update QUMA solution as described in Section 6.2. We test both Ferdinand and BROADCAST using the same evaluation methodology we describe in Chapter 4 and Sections 5.2.1 and 5.2.2, using both mixes of the TPC-W bookstore benchmark, in addition to the RUBiS auction and RUBBoS bulletin board benchmarks.

Figure 6.7 compares the performance of Ferdinand to our BROADCAST implementation. To ease comparison with our above results we've also included the SIMPLECACHE system with publish / subscribe-based consistency management.

For both mixes of the bookstore benchmark, BROADCAST's consistency management became a performance bottleneck before Ferdinand's cache miss rate otherwise limited system scalability. Even the

relatively read-heavy browsing mix contained enough updates to tax the consistency management system, and Ferdinand far outperformed the alternative approach. The comparison with SIMPLECACHE's performance is also notable. For the browsing mix, BROADCAST only modestly outperformed SIMPLECACHE, even though BROADCAST used DHT-based cooperative caching. For the shopping mix the update rate was sufficiently high that SIMPLECACHE even outperformed BROADCAST, confirming that consistency management can limit overall performance even for traditional query caching methods.

For the auction and bulletin board benchmarks the read-dominated nature of the workloads prevented consistency management from becoming the limiting factor. For these benchmarks, the system only processed about ten updates per second for even the scaled throughputs reached by SIMPLECACHE and Ferdinand, and update notifications were easily propagated in both consistency approaches.

This result demonstrates that attaining good scalability with query caching can require key improvements over both traditional caching and consistency management methods, showing that cooperative caching and publish / subscribe-based invalidation are both critical to Ferdinand's overall performance.

6.4 Conclusions

In this chapter we described Ferdinand's consistency management system. Ferdinand uses a scalable topic-based publish / subscribe system to efficiently communicate update notifications to each DBSS node. A key challenge in using publish / subscribe for consistency management is to map database queries and updates to subscriptions and publications they generate to ensure that a DBSS node will receive all updates that might affect a query result in its cache. We called this problem the Query / Update Multicast Association (QUMA) problem. We proved that given common assumptions about the publish / subscribe system, properties common to a typical modern DBMS, and a correct QUMA solution, Ferdinand's consistency management system will ensure that no stale query result is cached indefinitely at a Ferdinand node.

We showed how to use an offline analysis of the queries and updates of a Web application to determine their dependences and generate an efficient mapping between database requests to subscriptions and

publications, solving the QUMA problem Using our offline analysis, we generated two distinct QUMA solutions, and developed a DBSS publish / subscribe simulator to comparatively evaluate them. Our simulation results showed that both strategies far outperformed traditional broadcast-based consistency management, but that neither strategy dominated the other on all workloads. For peak consistency management efficiency, a DBSS administrator would need to evaluate a content provider's Web application on an application-by-application basis to determine which of the two strategies should be used.

Finally, we implemented our consistency management design within the Ferdinand and SIMPLECACHE DBSS's using one of the two QUMA solutions, and compared their performance to an Ferdinand-like DBSS that instead used broadcast-based consistency management. For some workloads, the broadcast-based consistency management limited the scalability of both Ferdinand and SIMPLECACHE, with a peak performance significantly below that attained by Ferdinand using publish / subscribe-based consistency management. For other workloads, however, even broadcast-based consistency management was sufficient to reach peak throughputs.

Overall, these results indicate that publish / subscribe can be used to efficiently implement consistency management for all of our workloads, and that for some of these workloads the consistency management implementation will be critical to maximizing a DBSS's scalability. For other workloads, however – those either limited by low cache hit rates or with relatively few updates – the consistency management system will not be the performance bottleneck of an Ferdinand-like DBSS even if simpler methods are used.

Chapter 7

Application-aware Query/Update Dependence Analysis

A key strength of the paradigm of database management systems (DMBSs) is that it encapsulates all data management tasks into a single infrastructure, separating those tasks from the applications where the data is used. Although this encapsulation is ideal from the perspective of good software engineering and programmatic design, it can cause inefficient performance because relationships among data are not maintained between the data's storage and use. Although some research has attempted to erase the barrier between the application and DBMS, most work to improve system performance focuses on just the DBMS or the database application individually, maintaining the separation imposed by the DBMS paradigm.

In many database applications the interactions between the application and DBMS are clearly specified at compile-time. This is especially true for Web applications, in which database requests are often written as templates with parameters that are filled in at run-time. For many Web applications and database request templates there is a clear relationship between the DBMS data and application data. For example, consider the Java application code in Figure 7.1. It selects a single value from the database and immediately sets an application variable (`x`) to that value.

Simple relationships between the DBMS data and application data enable a holistic analysis of the DBMS and the database application: we can maintain the relationships between the DBMS data and

```
public void salaryFoo(Connection connection, int id) {
    try {
        PreparedStatement namePs = connection.prepareStatement(
          "SELECT salary FROM emp WHERE id = ?");
        namePs.setInt(1, id)
        ResultSet nameResult = namePs.executeQuery();
        while (nameResult.next()) {
            int x = nameResult.get("salary");
            ...                                    // Foo!
        }
    } catch (SQLException e) {
    }
}
```

Figure 7.1: A Java code snippet to illustrate a simple relationship between DBMS and application data.

application data and optimize how the application uses the DBMS data while still maintaining their programmatic separation. As an example of the type of data relationships that can be maintained, consider a column within a relation that is specified to contain unique values. Values retrieved from this column are then known to be distinct, which can lead to better optimization of the application's execution.

Here we study only a very narrow use of these cross-boundary data relationships: we use them to better determine when the execution of a Web application is not affected by a database update, which we call the *application-update depedence* problem. This problem is especially important for Ferdinand: if the application is not affected by an update, then the Ferdinand nodes do not need to be notified of the update, reducing the amount of consistency traffic necessary.

In this chapter we describe techniques to maintain the relationship between the DBMS data and application data. We then apply these techniques to the problem of correctly executing Web applications while using database query result caches, showing that our holistic techniques outperform traditional techniques that focus just on the database requests or application alone. Our overall contributions are as follows:

- We introduce static, offline techniques that maintain simple known relationships between DBMS and application data.

- We apply the above techniques to find two holistic optimizations in our Web application benchmarks that cannot be found without holistic analysis.

- We use the holistic analysis-derived relationships to configure Ferdinand's consistency management system.

- We measure the contribution of our holistic analysis both statically and when used as part of Ferdinand, showing that holistic analysis outperforms the more traditional methods of query/update dependence analysis.

The remainder of this chapter is organized as follows. Section 7.1 describes how we maintain the relationship between database and application data for a database application. Sections 7.2 and 7.3 each describe a case in our Web application benchmarks for which holistic analysis can determine that a Web application is not affected by a database update when traditional analyses cannot make this determination. Section 7.4 applies the holistic analysis to the Web application benchmarks and shows that our holistic analysis improves on traditional query/update dependence analysis both statically and when used to configure Ferdinand's consistency management system.

7.1 Holistic analysis of database applications

To show that a database application is independent of a given update we extend the ideas of Blakeley et al. [20] and Elkan [37], showing that all data used by the application is independent of the update. The database requests in many applications consist of a small number of static templates within the application code. Typically, each template has a few parameters that are bound at run-time, with the template and its instantiated parameters defining the database request actually executed. The earlier techniques to show query-update independence extend easily to query and update templates. For many common query templates we can simply determine the data from which an instantiation of the template possibly derives, and for update templates we can similarly determine the data which an instantiation of the template possibly affects. By arguments analogous to those of Blakeley et al. and Elkan, the query

template is independent of the update template if no possible instantiated query can derive from data affected by a possible instantiated update. We applied this more traditional analysis in Chapter 6.

It is possible, however, for a query-update template pair to be dependent, but for an application using the query template to still be independent of any updates from the update template. This situation can occur in two ways. First, it is possible that although a query-update template pair is dependent, the application will never use template parameters that result in a dependent query-update pair. Second, it is possible that the application does use parameters that create a dependent query-update pair, but the application only uses a subset of the query result data that was not affected by the update. In both of these cases, determining the independence of the application from the update requires consideration of not just the database requests alone but also how the application issues and uses those database requests.

In our application-update dependence analysis, we refine the traditional approach by applying analytical techniques from optimizing compilers to determine how applications use their database request templates. In particular, we correlate application data to the database data from which it was derived. This allows us to propagate database metadata through the application and determine constraints on the parameter values actually used to instantiate database templates. All of our dependence analysis occurs offline, using just the static application code and its database request templates. We do not examine or modify the run-time execution of the application in any way.

The next two sections describe two classes of query-update template pairs for which application-update dependence analysis can determine that the application is not affected by the update but traditional analysis cannot. In Section 7.2 we describe the *unused-data* optimization, in which the query-update pair are dependent but the application does not use any affected data. Section 7.3 describes the *existing-primary-key* optimization, in which our compile time data analysis can determine that a run time parameter will always be an existing primary key for some relation, allowing us to prove that queries actually instantiated from that template will be independent of insertions to that relation.

7.2 The *unused-data* optimization

In some cases database application programmers retrieve more data from the database than they use in their application. This frequently occurs for requests that do not limit the projection of data retrieved from a row (i.e. "SELECT *"), and can also happen when either the application or database schema evolves and the programmer fails to update the database request to reflect the change. If a subsequent database update affects the unused data, that update will affect the database query result but not the application using the result. This section describes how we detect query result data that is unused by the database application, allowing us to infer that the application is not dependent on some updates that affect the query result.

To apply the *unused-data* optimization, we first use the database schema to expand any wildcard characters (*) in the projection clause to all the fields that are retrieved from the database by the query. We then examine the application's use of the query result set, marking each field in the query result as either used or unused by the application. Applications typically retrieve query result data using either the field name or the numerical index of the data they wish to access. For example, after executing the query "SELECT name, salary FROM emp WHERE id = 42" an application could retrieve the salary using either *getInt("salary")* or *getInt(2)*, so this process is usually straightforward. Sometimes our offline analysis can not determine which field of the result is being accessed, typically when the field is chosen using a variable set at runtime. In such cases we conservatively mark all fields as used to ensure the correctness of the optimization. We then remove any unused fields from the query and use our modified query in traditional query-update dependence analysis with each other update in the application, yielding a potentially finer analysis than was possible before. Note that we use our modified query only in our offline dependence analysis, and do not modify the query that is actually executed by the application at runtime. We do this so that we do not affect the performance characteristics of the application.

```
try {
    PreparedStatement namePs = connection.prepareStatement(
      "SELECT name FROM emp WHERE id = ?");
    ResultSet idsResult = statement.executeQuery(
      "SELECT id FROM emp WHERE salary > 10");
    while (idsResult.next()) {
        int empId = idsResult.getInt("id");
        namePs.setInt(1, empId)
        ResultSet nameResult = namePs.executeQuery();
        ...
    }
} catch (SQLException e) {
}
```

Figure 7.2: A Java code snippet to illustrate the *existing-primary-key* optimization.

7.3 The *existing-primary-key* optimization

The *exisiting-primary-key* optimization applies in situations where we can determine that a query result retrieves data based only on an existing primary key for some relation. In such a case, we then know that the query result is unaffected by insertions to that relation, since primary keys are guaranteed to be unique.

For example, let *emp(id, name, salary)* be a relation with primary key *id*. Consider the Java code snippet of Figure 7.2. This code first selects the *id* of all employees with salaries greater than 10, and then selects the name of each of employee based on those *id*s. If there are no intervening deletions or modifications to the primary keys of this relation, then each name query will retrieve the name of an employee already in the database, and thus the name query is unaffected by insertions to the employee relation. In this simple example the code snippet could be replaced by the single query "SELECT name FROM emp WHERE salary $>$ 10". In real applications similar interactions might be broken into multiple queries because the business logic is too complex to easily express as a single SQL query, or intervening user input might affect the application's execution.

To apply the *existing-primary-key* optimization we must first determine that a query's selection parameter is in fact a primary key for some relation, and then prove that the primary key has not been

modified or otherwise removed from the database. To accomplish the former we track the relationship between application data and database data, using data propagation techniques similar to those used when applying compiler optimizations. Whenever a variable is assigned a value originating from the database, we label the variable with the database metadata characteristics for the data. For example, in the code snippet above the line *int empId = idsResult.getInt("id");* would allow us to label the variable *empId* with the fact that its value originated from the *id* field of the employee relation. This fact would be propagated to when the value was used in the subsequent query (*namePs.setInt(1, empId)*) at which point the metadata can be used to refine the dependence analysis. This data propagation technique is well-suited to typical Java applications, because variables often have limited scope and direct memory access and pointer arithmetic are prohibited.

When the data source can be propagated to a query parameter we then check whether the data source was the primary key for this query's source relation and whether the parameter constrains the query to match just the row for the primary key. If this is the case, we examine all updates within the application to confirm that no update deletes rows from this relation and also that no update modifies the primary key data for the relation (primary keys are typically immutable). This allows us to conclude that the source data consists of a primary key that still exists in the relation, and thus that this query is unaffected by any insertions to the relation since primary keys are unique. As described, this technique can only succeed when the primary key for a relation is a single column, since we do not track the relationship between different variables in the application. Our technique could be modified to work for simple cases when the relation has a multi-column primary key, although such a modification might require more data to be tracked about each variable during the data propagation step.

7.4 Evaluating application-update dependence analysis

To evaluate the effectiveness of application-update dependence analysis we seek to answer two basic questions. First, to what extent do our improvements apply to existing database applications? Second, what performance advantages can be gained from applying them? To answer these questions we apply application-update dependence analysis to our three Web application benchmarks and compare the re-

	bookstore	auction	bulletin board
Total (no analysis)	464	308	507
Table-level analysis	85	82	88
Row- and column-level analysis	65	48	65
Row- and column-level plus *unused-data*	50	48	61
Row- and column-level plus *existing-primary-key*	54	41	50
Row- and column-level plus both methods	39	41	46

Table 7.1: Number of possibly dependent query-update template pairs for each benchmark application using various methods of dependence analysis.

sults to traditional query-update dependence analysis. For these applications we show that our methods can determine the independence of many query-update pairs whose relationship cannot be determined from traditional query-update dependence analysis. We then use the results of these dependence analyses as part of Ferdinand's consistency management infrastructure, and show that Ferdinand's consistency management requires significantly fewer messages with application-update-derived dependencies than with traditional dependence analysis.

This section is organized as follows. In Section 7.4.1 we apply both application-update dependence analysis and traditional query-update dependence analysis to the benchmarks, comparing the number of possible dependencies that can be ruled out with each technique. Section 7.4.2 examines the performance of application-update dependence analysis when its dependencies are used by Ferdinand's consistency management system.

7.4.1 Static gains of application-update dependence analysis

To evaluate the coverage of application-update dependence analysis we applied both traditional query-update dependence analysis and each of our dependence analysis improvements to each benchmark. For our evaluation we manually applied the optimizations to each benchmark, being careful to include only cases where we are certain that an automated implementation could perform the necessary analysis.

Table 7.1 shows the results of our comparison.

The bookstore benchmark contains 464 potential query-update template pairs. A coarse table-level dependence analysis determines that all but 85 query-update template pairs are independent. A finer-grained row- and column-level data analysis eliminates an additional 20 pairs, ruling just 65 to be possibly dependent. Of these 65 pairs, our application-update analysis determines that an additional 26 pairs are independent, an improvement of 40% over the traditional query-update dependence analysis. Of these 26 pairs, 15 possible dependencies were eliminated by our *unused-data* analysis: the query-update templates were possibly dependent, but the application did not actually use any of the possibly affected data. The other 11 pairs were ruled independent by our *existing-primary-key* analysis. Overall, our static analysis determines that 92% of the query-update template pairs are independent for the bookstore benchmark.

The auction benchmark contains 308 query-update template pairs. All but 82 pairs can be ruled independent by table-level analysis, and all but 48 independent by finer-grained analysis. Application-update dependence analysis finds only 7 additional query-update template pairs that are independent, a 14% improvement over the traditional data analyses. For this benchmark, the *unused-data* optimization is completely ineffective – the application always uses all data in a possibly-affected query – and all gains are from the *existing-primary-key* optimization. Our static analysis is not as good for the auction as for the bookstore benchmark; here it only eliminates 87% of the template pairs as independent.

The bulletin board benchmark contains 507 query-update template pairs, and all but 88 of these can be ruled independent by table-level data analysis. Row- and column-level data analysis determines that 65 of these pairs may be dependent, and application-level analysis determines that 19 of those 65 pairs are surely independent, a 29% improvement over the traditional analysis. Of these 19, 4 were found by the *unused-data* optimization and 15 were found by the *existing-primary-key* optimization. Our static analysis eliminates about 91% of the template pairs as independent for this benchmark.

These results demonstrate that the effectiveness of the *unused-data* optimization can vary significantly between applications. For the auction benchmark, all queries are written to retrieve specific columns for rows that match some criteria, and all retrieved information is immediately displayed or used in

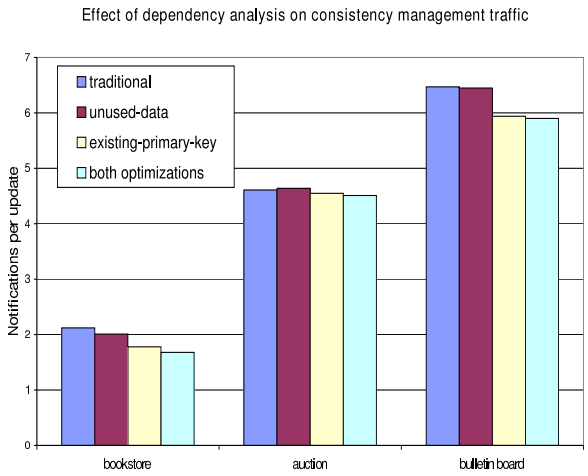


Figure 7.3: Number of notification messages per update for each benchmark and each dependence analysis.

the auction's business logic. The bookstore and bulletin board benchmarks, however, both sometimes retrieve data that is not used by the application. In all of these cases, all columns of a row (or set of rows) are retrieved but only some columns are used in the subsequent computation – and any possibly affected columns are not used by the application.

For applications like these, the *existing-primary-key* optimization is highly effective. All of these applications contain interactions where a set of rows is retrieved from the database, the application implements some business logic based on the retrieved data, and additional queries are executed using data from the earlier result set. In many of these cases, our analysis can determine that the subsequent query depends only on data previously existing in the database, and therefore that the query is not affected by insertions to the relevant table. We expect interactions like this to be common, and thus expect that the *existing-primary-key* optimization will apply to a wide range of Web applications.

7.4.2 Dynamic gains of application-update dependence analysis

To evaluate the gains of application-update dependence analysis in an applied setting, we compared the performance of Ferdinand's consistency management system using traditional query-update dependence analysis to the performance when using our methods. For the bookstore and auction benchmarks we

executed the application on 8 Ferdinand nodes for 15 minutes at approximately 50% system utilization, starting with warm query result caches, and measured the number of update notifications required by the consistency management system. For the bulletin board benchmark we did the same, but using a Ferdinand system with 12 nodes. The results of this evaluation are in Figure 7.3.

For both the bookstore and bulletin board benchmarks, the application-update-derived dependencies reduced consistency management traffic by 10% compared to query-update dependence analysis. The improvement for the auction benchmark is more slight. From the experiments using just the *unused-data* or *existing-primary-key* optimizations, we see that nearly all performance gains come from the *existing-primary-key* optimization. This result is true for the bookstore and bulletin board benchmarks even though the *unused-data* optimization identified many query-update template pairs as independent, when the pairs' relationships could not be determined using traditional dependence analysis or the *existing-primary-key* analysis. This fact indicates that simply measuring the number of dependent query-update template pairs is not sufficient to discern real performance gains, and that any meaningful performance analysis must consider the dynamic execution of the database application. Determining the independence of a frequently executed query or update is much more valuable than determining the independence of a rare database request.

Overall, the value of the *unused-data* optimization is unclear. Its applicability is more variable than that of the *existing-primary-key* optimization, and even when *unused-data* determines that query-update template pairs are independent, for our tested benchmarks those queries and updates seem uncommon in practice. For our benchmarks the *unused-data* optimization most commonly applies to join queries when the application requires all columns from one relation but only several columns from another relation. In such a case a typical programmer might project all columns of the result set (i.e. "`SELECT *`") rather than specify just the needed columns, since all columns are used from one of the relations. If this is the dominant case where the *unused-data* optimization applies, then it is not surprising that its overall effect on performance is slight since join queries are executed less frequently than simpler retrieval and update queries for most typical Web application workloads.

The *existing-primary-key* optimization produced significant gains for all workloads, with the gains

relative to its applicability in the workload. We expect this optimization to apply to most Web applications, as it identifies a common data usage scenario: because it applies to simple insertion and row retrieval queries, we expect it to result in similar performance gains for a wide class of applications.

7.5 Conclusions

In many environments database requests are executed as part of a client database application. In this chapter, we extend the techniques of query-update dependence analysis to instead determine the dependencies between database updates and the applications that execute the database queries. We show how the application and database requests can be analyzed together to infer relationships that cannot otherwise be determined using traditional query-update dependence analysis, describing two specific cases where we improve upon earlier dependence analysis. In the first we show how one can sometimes determine that an application is not affected by a database update even though the application uses a query result affected by the update, in cases when the query result is not wholly used by the application. Our second improvement shows how we can determine that some query results are not affected by insertions to the query's relation, by proving that the query result depends only on data already existing in the relation. We then applied our application-update dependence analysis to our three benchmark Web applications, showing that our techniques are not merely theoretical but are likely to apply to real database applications. Finally, we executed those Web applications using Ferdinand, showing how our application-update dependence analysis required less communication to maintain cache coherence than when traditional query-update dependence analysis was used.

In our latter optimization we introduced a powerful new technique of tracking database metadata across the application-database boundary. Here we used this technique specifically to determine when parameters in a database query originated from the database via an earlier query, allowing us to infer the independence from insertions described above. Tracking the relationship between application data and database data, however, is a new general tool that might yield advancements in many database subfields; we plan to explore this technique further.

Chapter 8

Conclusions

With a growing population of Internet users and a continuing shift toward increasingly complex, interactive online experiences, user demand for Web applications will continue to increase at a fast pace in the foreseeable future. Some of this demand can be alleviated with improvements in computer hardware and further hardware infrastructural investments, but (1) innovation in hardware technology is unlikely to match the pace of user demand growth, and (2) large investments in permanent infrastructure are not economically feasible for many content providers, especially content providers who experience widely varying or unpredictable loads.

Along with [65], this dissertation is one of the first detailed explorations of providing database scalability as an economically efficient third-party service to alleviate the database bottleneck for Web applications. We have put forth the thesis that significant database scalability can be achieved by a third-party service while meeting the privacy and quality of service requirements of a content provider. The approach taken here was largely to assume the architectural constraints imposed by those privacy and quality of service requirements (as given in [65]), and show that scalability is still attainable. Our work is based on a blend of justifying our design with abstract and theoretical arguments and validating it with a full, working implementation of a prototype database scalability service, Ferdinand.

The primary novel aspects of Ferdinand's design are its multi-tiered cooperative query result cache, its use of publish / subscribe for cache consistency management, and how we use an offline analysis of

a content provider's Web application to make consistency management more efficient. In the next three sections we continue by discussing the impact and limitations of our work for each of these aspects, respectively. We finally conclude with some closing remarks in Section 8.4.

8.1 Ferdinand's query result cache

There is a bit of a divide in the Web scalability community as to what is the right way to cache data to provide scalability for Web applications. At one end of the spectrum are proponents of output-level caching, caching the whole pages and page fragments generated by the Web application. Other projects take the diametrically opposite approach, replicating or caching the raw data and implementing query processing on that data at each server node. Each of these approaches has some advantages. For output-level caching, any data reuse scales not just the database component but also eliminates work at the application server. The latter approach, though, has the advantage that it conforms to a Web programmer's standard model of how content is generated and is more general, also working for applications that have non-output-related side effects.

By caching database data as materialized query results and not at the page level or as raw data, Ferdinand takes a middling approach that has some of the advantages of both philosophies. Like output-level caching, Ferdinand eliminates load at the database server, but unlike raw-data-caching does not create new load by introducing complex query processing at a DBSS node. Ferdinand does not eliminate work at the application server, however, but conforms to the standard Web application model as a Web programmer might expect.

Until now, no previous work has shown that caching of database query results in this form can provide significant scaling of Web applications. Much of our scalability comes from our multi-tier cache design, which enables very high scalability for frequently used query results and still only requires a single execution of a query at the database server for the whole DBSS each time that query result is updated. Our design has the additional advantage that it is compatible with content providers that require privacy guarantees from a DBSS: Ferdinand's cached query results can be encrypted, allowing

scalability without compromising data privacy in some circumstances. By validating our approach with a performance analysis of a real implementation, our work leaves no doubt that query result caching is a feasible approach for at least some Web applications.

One of the biggest limitations of our work, however, is that the multi-tiered cache can increase the latency of Web interactions, especially when the content provider's home database server is far from the DBSS nodes. In [65], Manjhi addresses this concern for some Web applications, showing how a DBSS can reduce the interactions between the DBSS nodes and the home database server, reducing the overall latency of the Web interaction. Also, our evaluation shows that Ferdinand significantly scales Web applications while meeting latency requirements even in medium-latency wide-area environments. This suggests that a DBSS could be implemented as a semi-centralized shared, third-party service. In this setting, a DBSS might maintain multiple medium-sized installations at different data centers, geographically distributed but each near to the Internet core. Each content provider would be assigned to use the DBSS installation nearest to their home server, which could then provide economically efficient scalability services on their shared infrastructure without significantly increasing the overall latency of each Web interaction.

Finally, the scalability of our overall design is limited by our requirement that all database updates be forwarded to the home database server, with affected queries being later re-executed at the home server. Alternative designs might use our multi-tiered query result cache but avoid forwarding all updates. Such an approach would have the disadvantage that the content provider might not maintain full, centralized ownership of their data, but this problem might be acceptable for some types of data, e.g. a temporary shopping cart. A middle ground that we did not explore was to instead immediately propagate updates within the query result caches, rather than require re-execution of the queries at the home database server.

8.2 Publish / subscribe for consistency management

Within the database lore, it is generally accepted that broadcast-based consistency management is not scalable and therefore not suitable for systems that must support high loads, but there is a surprising

dearth of real data to support this claim. To the best of our knowledge, we are the first to show that broadcast-based methods are the scalability bottleneck for a caching system when all updates are sent to the home database server. Even for our evaluation, however, the broadcast-based method was the bottleneck for just one application, the TPC-W bookstore benchmark. To some extent, the disparity between the lore and real data might be due to the general lack of scalability of database servers. Even our SIMPLECACHE DBSS implementation (using our publish / subscribe-based consistency management) only slightly out-scaled a broadcast-based system, showing that the consistency management would likely have been perceived as only a slight bottleneck for earlier implementations. We predict that consistency management will be considered to be a more significant bottleneck as database systems scale even further.

To use publish / subscribe for Ferdinand's consistency management, we had to overcome a significant gap between the simple matching operation supported by scalable publish / subscribe systems, and the complex matching between database updates and the queries they affect. We did this by using an offline analysis of each Web application to constrain the range of updates and queries that the publish / subscribe system would have to support, essentially moving much of the matching process to be external to the publish / subscribe system. Our evaluation shows that this approach is very effective for applications where there is a fine relationship between updates and the queries they affect. For the TPC-W bookstore, our publish / subscribe-based system required an extremely small number of update notifications even for very large DBSS systems. Our evaluation is limited, however, by the small number of applications for which we implemented consistency management. At this time even our own endorsement of this technique must be tentative: to gain full support, publish / subscribe-based consistency management must be implemented and evaluated for a much wider range of applications.

One additional concern that this dissertation did not address is the possibility of ad-hoc updates in the database workload. In the worst case, one possible solution would be to invalidate all cached query results any time an ad-hoc update is issued. A more balanced approach, however, would be to simply broadcast ah-hoc updates to every DBSS node, which should be feasible for even medium- to large-sized DBSS installations if ad-hoc updates are only a small part of the workload as we typically expect.

8.3 Application-aware dependence analysis

The final major contribution of this dissertation is the conceptual contribution of using compiler-like analysis of a database application to better understand the dependence of the application on the database data. Fully understanding this relationship is a long-standing database problem, and most other approaches have required the application programmer to either explicitly declare this relationship or use tools designed to clarify it. In contrast, we have taken the approach that the programmer should write the application using whatever tools are best suited to their application development methodology, and we should use external automated tools to understand the application-database relationship as best as we can given this constraint.

Specifically, we used compiler-like analysis to more accurately configure our publish / subscribe system for consistency management within Ferdinand, evaluating two specific cases where traditional query / update dependence analysis could not accurately determine the relationship between the application and the database data, but a holistic analysis could. We showed that our holistic methods reduced consistency traffic by about 10% over methods that did not consider the application's use of the database requests. It is notable, however, that our evaluation of even these two techniques is limited. We report a reduction of traffic by about 10%, but the real question is what fraction of *unnecessary* consistency traffic did we eliminate, a value that must be greater than just 10%. Our evaluation is also limited by our failure to examine the real performance contribution of better consistency management on the throughput of our Web applications. For the RUBiS auction and RUBBoS bulletin board this contribution is clearly negligible since consistency management is not the bottleneck for any of those workloads. For the TPC-W bookstore, however, better consistency management might conceivably result in better overall system performance.

A major limitation of our work here is that our core techniques are specific to implementations of Web applications that rely on SQL requests embedded in application code. Our approach, however, is easily generalizable to a wider variety of methods of Web application programming. In some cases (e.g., when a programmer uses a persistence framework), the information available to an external data analysis is even greater than with embedded SQL requests, potentially allowing better understanding

of the dependence between an application and its database data. A problem with this line of work is that the technologies used in Web application development are themselves not yet mature. In the course of even a small number of years, application programmers have shifted from primarily using PHP and Java toward more diverse technologies including JSP, ASP, JavaScript, and more recently even source-to-source compilers like the Google Web Toolkit. We feel that our work, however, has validated the general approach of using a holistic data analysis of Web applications and their database requests to refine consistency management for a database scalability system.

8.4 Closing remarks

With increasing demand for Web-based services, scaling Web applications will continue to be a problem for the foreseeable future. This dissertation has addressed the problem of providing scalability for a relatively small class of Web applications: those that require up-to-date data, but do not require full multi-statement transactional consistency for most operations.

As time has progressed, there has been an increasing trend toward the decomposition of traditional Web applications into focused Web services. In many ways, this trend has increased the relevance of our work: even in applications where strong transactional consistency is traditionally needed – e.g., banking, online sales, etc. – only a small number of the decomposed services require strong consistency. For the other parts of the service, a system like Ferdinand could be used to provide economical scalability without violating consistency or privacy constraints.

Many aspects of our evaluation are limited to the technologies of our time. However, our overall approach will likely generalize to the future of Web application programming. One of the strengths of our approach is that we rely on simple, well-understood techniques – such as distributed hash tables, publish / subscribe, and a simple representation of query results as materialized views – to solve the underlying scalability problem. As Web programming evolves and more complex interactions such as application services become more common, many of our techniques will continue to apply to the challenge of scaling the content provider's application.

Appendix A

The Algorithms of the SIMPLECACHE DBSS

SIMPLECACHE's algorithms highly resemble those of Ferdinand with two main differences: because there is no distributed cooperative cache, SIMPLECACHE does not need to handle database queries at a master DBSS node or manage the consistency of a distributed cache. This leaves SIMPLECACHE with only three main algorithms: (1) for a database query at a SIMPLECACHE node, (2) for a database update at a SIMPLECACHE node, and (3) for handling an update notification on a publish / subscribe topic.

SIMPLECACHE's functions and data structures are also similar to those of Ferdinand, and the functions and data structures used in our algorithms below refer to the same as those described in Tables 3.1 and 3.2.

We continue by discussing each of SIMPLECACHE's algorithms in turn.

A.1 Processing a query at a SIMPLECACHE node

Algorithm A.1 shows the algorithm for processing a database query at a SIMPLECACHE node. The SIMPLECACHE node first checks for the query result in its cache. If the query result is present in the cache, the node immediately replies with the result, completing the interaction. Otherwise, it proceeds much like the master node in Ferdinand: it computes the topics associated with the query (rather than the master topics), subscribes to those topics and waits for confirmation, and forwards the query to the

Algorithm A.1 Processing a query at a SIMPLECACHE node. As input, this algorithm takes a database query Q.

```
01  R ← cacheMap.get(Q)
02  if (R) return R
03  pendingAndValid.add(Q)
04  topics ←computeTopics(Q)
05  foreach t ∈ topics
06        topicToQueryMap.add(t, Q)
07        subscribe(t)
08        waitForReply()
09  R ←homeServerSend(Q)
10  if pendingAndValid.contains(Q)
11        cacheMap.add(Q, R)
12        pendingAndValid.remove(Q)
13  return R
```

Algorithm A.2 Processing an update at a SIMPLECACHE node. As input, this algorithm takes a database update U.

```
01  R ←homeServerSend(U)
02  waitForReply()
03  topics ←computeTopics(U)
04  foreach t ∈ topics
05        publish(t,U)
06  return R
```

home database server. When the query result is received it likewise checks for intervening updates and places the query result in its cache if possible. It finally returns the query result to the application server.

A.2 Processing an update at a SIMPLECACHE node

Algorithm A.2 shows the algorithm for processing a database update at a SIMPLECACHE node. When an update is received, it is immediately forwarded to the home database server. When the home database server replies, the consistency manager computes all publish / subscribe topics associated with that update and publishes the update to those topics. The SIMPLECACHE node finally replies with confirmation of the update.

A.3 Handling an update notification at a SIMPLECACHE node

Algorithm A.3 shows the algorithm for processing an update notification at a SIMPLECACHE node. The SIMPLECACHE node uses its topic-to-query map to determine any possibly-affected queries and removes query results that it cannot determine to be unaffected by the update. It then marks any affected pending queries as invalid and unsubscribes to any topics on which no cached queries depend.

Algorithm A.3 Handling an update notification at a SIMPLECACHE node. As input, this algorithm takes the published update U and the topic T on which the update notification was received.

```
01  queries ← topicToQueryMap.get(T)
02  foreach q ∈ queries
03      if possiblyAffects(U, q)
04          pendingAndValid.remove(q)
05          cacheMap.remove(q)
06          reducedTopics ←computeTopics(q)
07          foreach t ∈ reducedTopics
08              topicToQueryMap.remove(t, q)
09              if (topicToQueryMap.get(t) = ∅)
10                  unsubscribe(t)
```

Bibliography

[1] Adobe Flash. http://www.adobe.com/products/flash/.

[2] Akamai Technologies, Inc. http://www.akamai.com.

[3] Akamai Technologies, Inc. Akamai edgesuite. http://www.akamai.com/html/services/edgesuite.html.

[4] Akamai Technologies, Inc. Logitech implements Akamai edge computing. http://www.akamai.com/en/resources/pdf/casestudy/Akamai_CaseStudy_Logitech.pdf.

[5] Gustavo Alonso. Partial database replication and group communication primitives. In *Proc. European Research Seminar on Advances in Distributed Systems*, 1997.

[6] Peter Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE*, pages 562–570, 1976.

[7] Amazon.com, Inc. http://www.amazon.com.

[8] Amazon.com, Inc. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[9] Amazon.com, Inc. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[10] Amazon.com, Inc. Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[11] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Dbproxy: A dynamic data cache for web applications. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 821–831. IEEE Computer Society, 2003.

[12] Khalil Amiri, Sara Sprenkle, Renu Tewari, and Sriram Padmanabhan. Exploiting templates to scale consistency maintenance in edge database caches. In *Proc. International Workshop on Web Content Caching and Distribution*, 2003.

[13] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *ICDE*, pages 230–241. IEEE Computer Society, 2005.

[14] Cristiana Amza, Gokul Soundararajan, and Emmanuel Cecchet. Transparent caching with strong consistency in dynamic content web sites. In Arvind and Larry Rudolph, editors, *ICS*, pages 264–273. ACM, 2005.

[15] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A model for concurrency in nested transactions systems. *J. ACM*, 36(2):230–269, 1989.

[16] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 599–610, New York, NY, USA, 2006. ACM.

[17] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *PODC*, pages 114–122, 1983.

[18] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[19] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Trans. Software Eng.*, 5(3):203–216, 1979.

[20] José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[21] Andre B. Bondi and Ward Whitt. The influence of service-time variability in a closed network of queues. *Perform. Eval.*, 6(3):219–234, 1986.

[22] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. W3C web services architecture. http://www.w3.org/TR/ws-arch/, 2004.

[23] Erik Brynjolfsson, Michal D. Smith, and Yu Jeffrey Hu. Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. 2003. MIT Sloan Working paper No. 4305-03, 2003.

[24] CacheFly. http://www.cachefly.com.

[25] California Senate. Bill SB 1386. http://info.sen.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_1386_bill_20020926_chaptered.html, 2002.

[26] Ikram Chabbouh and Mesaac Makpangou. Caching dynamic content with automatic fragmentation. In Gabriele Kotsis, David Taniar, Stéphane Bressan, Ismail Khalil Ibrahim, and Salimah Mokhtar, editors, *iiWAS*, volume 196 of *books@ocg.at*, pages 975–986. Austrian Computer Society, 2005.

[27] Ikram Chabbouh and Mesaac Makpangou. FRACS: A hybrid fragmentation-based CDN for e-commerce applications. 2007.

[28] Jim Challenger, Paul Dantzig, Arun Iyengar, and Karen Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Trans. Internet Techn.*, 5(2):359–389, 2005.

[29] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM*, pages 294–303, 1999.

[30] Badrish Chandramouli, Junyi Xie, and Jun Yang. On the database/network interface in large-scale publish/subscribe systems. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 587–598. ACM, 2006.

[31] Chun Yi Choi and Qiong Luo. Template-based runtime invalidation for database-generated web contents. In *APWeb*, volume 3007 of *Lecture Notes in Computer Science*, pages 755–764. Springer, 2004.

[32] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[33] Edith Cohen, Eran Halperin, and Haim Kaplan. Performance aspects of distributed caches using ttl-based consistency. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 744–756. Springer, 2001.

[34] James R. Cordy, Anatol W. Kark, and Darlene A. Stewart, editors. *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative Research, October 17-20, 2005, Toronto, Ontario, Canada*. IBM, 2005.

[35] Eric Dash and Tom Zeller Jr. Mastercard says 40 million files put at risk. *The New York Times*, June 18, 2005.

[36] eBay, Inc. http://www.ebay.com.

[37] Charles Elkan. Independence of logic database queries and updates. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee*, pages 154–160. ACM Press, 1990.

[38] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[39] Sergei Evdokimov, Matthias Fischmann, and Oliver Gunther. Provable security for outsourcing database operations. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 117, Washington, DC, USA, 2006. IEEE Computer Society.

[40] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, rfc 2616. ftp://ftp.isi.edu/in-notes/rfc2616.txt, June 1999.

[41] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 70–75, New York, NY, USA, 1982. ACM.

[42] Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*. ACM, 2002.

[43] Lei Gao, Michael Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Improving availability and performance with application-specific data replication. *IEEE Trans. Knowl. Data Eng.*, 17(1):106–120, 2005.

[44] James Gosling and Henry McGilton. The java language environment: A white paper. http://java.sun.com/docs/white/langenv/, May 1996.

[45] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *VLDB*, pages 144–154. IEEE Computer Society, 1981.

[46] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 173–182. ACM Press, 1996.

[47] Jim Gray, Pete Homan, Henry F. Korth, and Ron Obermarck. A straw man analysis of the probability of waiting and deadlock in a database system. In *Berkeley Workshop*, page 125, 1981.

[48] Tobias Groothuyse, Swaminathan Sivasubramanian, and Guillaume Pierre. GlobeTP: template-based database replication for scalable web applications. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 301–310. ACM, 2007.

[49] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *9th International Conference on Database Systems for Advanced Applications*, 2004.

[50] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In Franklin et al. [42], pages 216–227.

[51] Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. In *ICDE*, pages 29–. IEEE Computer Society, 2002.

[52] IBM. IBM DB2 Database Server. http://www.ibm.com/db2/.

[53] Akamai Technologies Inc. and Jupiter Research Inc. Akamai and Jupiter Research identify '4 seconds' as the new threshold of acceptability for retail web page response times. http://www.akamai.com/html/about/press/releases/2006/press_110606.html.

[54] Akamai Technologies Inc. and Quocirca. Akamai and quocirca identify '4 second' performance threshold for european web-based enterprise applications. http://www.edgejava.net/html/about/press/releases/2007/press_110707.html.

[55] Robert O'Harrow Jr. Advertiser charged in massive database theft. *The Washington Post*, July 22, 2004.

[56] Alexandros Labrinidis and Nick Roussopoulos. Update propagation strategies for improving the quality of data on the web. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 391–400. Morgan Kaufmann, 2001.

[57] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *ICDE*, pages 177–189. IEEE Computer Society, 2004.

[58] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *VLDB*, pages 171–181. Morgan Kaufmann, 1993.

[59] Wen-Syan Li, Oliver Po, Wang-Pin Hsiung, K. Selçuk Candan, and Divyakant Agrawal. Freshness-driven adaptive caching for dynamic content. In *DASFAA*, pages 203–. IEEE Computer Society, 2003.

[60] Wen-Syan Li, Oliver Po, Wang-Pin Hsiung, K. Selçuk Candan, Divyakant Agrawal, Yusuf Akca, and Kunihiro Taniguchi. Cacheportal ii: Acceleration of very large scale data center-hosted database-driven web applications. In *VLDB*, pages 1109–1112, 2003.

[61] Limelight Networks. http://www.limelightnetworks.com.

[62] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In Franklin et al. [42], pages 600–611.

[63] Kaloian Manassiev and Cristiana Amza. Scalable database replication through dynamic multiversioning. In Cordy et al. [34], pages 141–154.

[64] Kaloian Manassiev and Cristiana Amza. Scaling and continuous availability in database server clusters through multiversion replication. In *DSN*, pages 666–676. IEEE Computer Society, 2007.

[65] Amit Manjhi. *Increasing the Scalability of Dynamic Web Applications*. PhD thesis, Carnegie Mellon University, Computer Science Department, March 2008.

[66] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 461–472. ACM, 2007.

[67] Microsoft. Microsoft SQL Server. http://www.microsoft.com/sql/.

[68] Microsoft Conference Management Toolkit. http://cmt.research.microsoft.com/cmt/.

[69] MySQL AB. MySQL database server. http://www.mysql.com.

[70] National Institute of Standards and Technology. Secure hash standard, federal information processing standards publication 180-2. `http://csrc.nist.gov/publications`, August 2002.

[71] ObjectWeb Consortium. Rice University bidding system. http://rubis.objectweb.org/.

[72] ObjectWeb Consortium. Rice University bulletin board system. http://jmob.objectweb.org/rubbos.html.

[73] Oracle. Oracle fusion middleware: Oracle toplink. http://www.oracle.com/technology/products/ias/toplink/.

[74] Panther Express. http://www.pantherexpress.com.

[75] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[76] Craig Partridge, Paul Barford, David D. Clark, Sean Donelan, Vern Paxson, Jennifer Rexford, and Mary K. Vernon. *The Internet Under Crisis Conditions: Learning from September 11*. National Academy Press, Washington, DC, January 2003.

[77] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[78] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCS Workshops*, pages 611–618, 2002.

[79] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, 32(3):241–280, 1999.

[80] The Associated Press. Credit card breach raises broad concerns. *The New York Times*, March 23, 2008.

[81] Red Hat Middleware, LLC. Hibernate: Relational persistence for java and .net. http://www.hibernate.org/.

[82] Ron Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computations*, pages 169–179. Academic Press, 1978.

[83] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[84] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2001.

[85] Peter Schatte. The M/GI/1 queue as limit of closed queueing systems. *Math. Operationsforsh. u. Statist. ser. Optimization*, 15:161–165, 1984.

[86] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*. USENIX, 2006.

[87] Robert M. Shapiro and Robert E. Millstein. Reliability and fault recovery in distributed processing. In *OCEANS*, volume 9, pages 425–429, September 1977.

[88] Swaminathan Sivasubramanian, Gustavo Alonso, Guillaume Pierre, and Maarten van Steen. Globedb: autonomic data replication for web applications. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 33–42. ACM, 2005.

[89] Gokul Soundararajan and Cristiana Amza. Online data migration for autonomic provisioning of databases in dynamic content web servers. In Cordy et al. [34], pages 268–282.

[90] Gokul Soundararajan and Cristiana Amza. Using semantic information to improve transparent query caching for dynamic content web sites. In *DEEC*, pages 132–138. IEEE Computer Society, 2005.

[91] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In Yolande Berbers and Willy Zwaenepoel, editors, *EuroSys*, pages 89–102. ACM, 2006.

[92] Michael Stonebreaker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, May 1979.

[93] The PHP Group. Php: Hypertext preprocessor. http://www.php.net.

[94] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *16th IEEE Computer Society International Conference*, pages 55–62, Spring 1978.

[95] Transaction Processing Council. TPC-W specification ver. 1.7. http://www.tpc.org/tpcw/.

[96] Gero M. Uhl. Large-scale content-based publish / subscribe systems. Ph.D. thesis, Darmstadt University of Technology, 2002.

[97] Zhou Wei, Dejun Jiang, Guillaume Pierre, Chi-Hung Chi, and Maarten van Steen. Service-oriented data denormalization for scalable web applications. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *WWW*, pages 267–276. ACM, 2008.

[98] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.

[99] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.