Scalable Query Result Caching for Web Applications

Charles Garrod
charlie@cs.cmu.edu
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213

Amit Manjhi
amitmanjhi@google.com
Google, Inc.

Anastasia Ailamaki
natassa@cs.cmu.edu
Carnegie Mellon University
EPFL

Bruce Maggs
bmm@cs.cmu.edu
Carnegie Mellon University
Akamai Technologies

Todd Mowry
tcm@cs.cmu.edu
Carnegie Mellon University
Intel Research Pittsburgh

Christopher Olston
olston@yahoo inc.com
Yahoo! Research

Anthony Tomasic
tomasic+@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

The backend database system is often the performance bottleneck when running web applications. A common approach to scale the database component is query result caching, but it faces the challenge of maintaining a high cache hit rate while efficiently ensuring cache consistency as the database is updated. In this paper we introduce Ferdinand, the first proxy-based cooperative query result cache with fully distributed consistency management. To maintain a high cache hit rate, Ferdinand uses both a local query result cache on each proxy server and a distributed cache. Consistency management is implemented with a highly scalable publish / subscribe system. We implement a fully functioning Ferdinand prototype and evaluate its performance compared to several alternative query-caching approaches, showing that our high cache hit rate and consistency management are both critical for Ferdinand's performance gains over existing systems.

1. INTRODUCTION

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 attacks. Content Distribution Network (CDN) provides a scalable solution for the delivery of *static* content by using a large shared infrastructure of proxy servers to absorb load spikes. CDN infrastructure places proxy servers near the end user to provide good performance; since static content changes very slowly, updates do not pose a significant problem for maintaining data consistency. *Dynamic* content, however, is generated in real-time and is typically customized for each user. An application runs code (such as a Java servlet) that makes queries to a backend database to customize the content, typically based on the user's request or a stored user profile. Off-loading the work of the application server (e.g., executing the Java servlets) to proxy nodes is not difficult, but the central database server remains a performance bottleneck [22].

Traditionally, replication of the database among several servers removes the performance bottleneck of a centralized database. However, this solution requires low-latency communication protocols between servers to implement update consistency protocols. Thus, low inter-server latency for update consistency directly conflicts with low user-server latencies demanded by a global infrastructure. In addition, database performance typically does not scale linearly with its cost, so provisioning for load spikes can be expensive.

Recently, a number of systems have been proposed with a similar architecture for scaling the delivery of database-backed dynamic content [3, 2, 17, 22]. In each of these systems users interact with proxy servers that mimic a traditional three-tiered architecture (containing a web server to handle user requests, an application server to generate dynamic content, and a database server as a backend data repository). These proxy servers typically cache static content and generate dynamic content locally, using a database query result cache and forwarding requests to the backend database server as needed. These design decisions successfully remove the application server load from the centralized infrastructure and reduce the number of requests sent to the backend database server, achieving increased scalability over systems that do not perform database query result caching. However, performance of these systems is limited by two key factors. First, most of these systems inefficiently maintain the consistency of the database query caches. Thus, their performance is hindered for workloads that require many updates to the database. Second, for many workloads, the overall scalability of these systems is limited by low cache

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

hit rates.

In this paper we propose Ferdinand, a proxy-based database caching system with a scalable, fully distributed consistency management infrastructure. In Ferdinand the proxy servers cooperate with each other to shield queries from the centralized database. If a database query result is not present in one proxy's cache, the result might still be obtained from another proxy server rather than from the central database. Ferdinand maintains cache consistency using publish / subscribe as a communications primitive. In our design the backend database system tracks no information about the contents of each proxy cache and performs no extra work to maintain their consistency, effectively lifting the burden of consistency management off the centralized server.

We envision a number of scenarios in which Ferdinand might be used. A business might temporarily employ Ferdinand on its local network to cost-effectively provision for higher load during a promotion, or a growing company might use Ferdinand to delay a costly upgrade to their database system. Alternatively, a CDN-like company might offer Ferdinand as a subscription service, scaling the web application and backend database for its customers, the content providers. Our intent is for Ferdinand to improve the price-to-performance ratio in situations where additional database scalability is needed. To be practical for these scenarios Ferdinand is as transparent as possible to the web application and central database server. We do not require the web designer or application programmer to learn a new methodology or optimize their code for execution on Ferdinand. Indeed, the only change necessary to run an application on Ferdinand is to replace the application's original database driver with our own.

The contributions of this paper are:

- We introduce Ferdinand, a new cooperative proxy architecture for dynamic content delivery, using distributed database query result caching and scalable consistency maintenance.
- We describe offline techniques to analyze a web application and its database requests, showing how to efficiently use topic-based publish / subscribe to maintain the consistency of database query result caches.
- We implemented a complete Ferdinand prototype consisting of a JDBC driver, local and remote caches and cache policies, and a cache invalidation mechanism. The prototype uses industry standard publish / subscribe and DHT implementations.
- We evaluated the prototype by executing standard benchmarks on the Emulab testbed [27] and measuring results with standard metrics. Our prototype scales dynamic content delivery for these benchmarks by as much as a factor of 10 over non-caching systems and attains as much as 3 times the throughput of traditional query-caching implementations.

In Section 2 we describe the architecture of the Ferdinand system. Section 3 discusses the problem of efficiently mapping a web application's database requests into our scalable consistency management infrastructure and presents our approach for doing so. Section 4 describes our prototype implementation of Ferdinand, which we evaluate in Section 5.

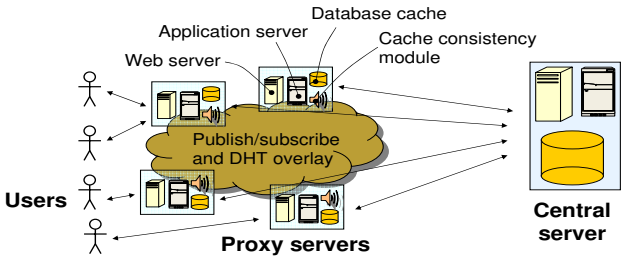

Figure 1: High-level architecture of Ferdinand

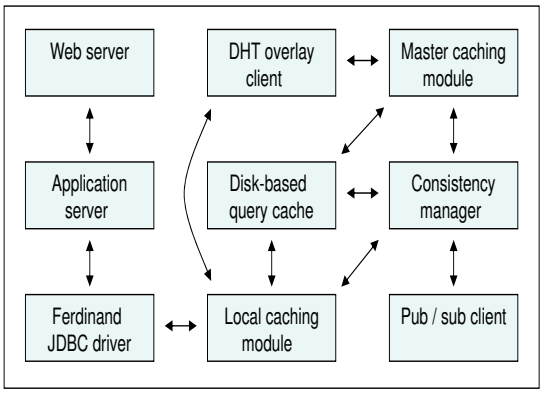

Figure 2: Component diagram of a Ferdinand proxy server

Section 6 discusses related work and Section 7 concludes and discusses future work.

2. THE FERDINAND ARCHITECTURE

This section describes the architecture of Ferdinand, including the design of the cooperative caching system and a high-level overview of its consistency management infrastructure.

The architecture of Ferdinand is shown in Figure 1. As with other proxy architectures, Ferdinand users connect directly to proxy servers rather than the centralized home server. Each proxy server consists of a static web cache, an application server, a database cache, and a cache consistency module. Unlike other architectures, Ferdinand proxy servers communicate with each other using a publish / subscribe infrastructure, and each proxy is also a member of a Distributed Hash Table (DHT) overlay.

The diagram in Figure 2 shows the interaction between components of a Ferdinand proxy server. The cache at each proxy is a map between each database query and a materialized view of that query's result. When a dynamic web application issues a database query, the proxy's local caching module responds immediately using the query result cache if possible. If the database query result is not present in the cache the DHT module hashes and forwards the query to the appropriate proxy server (the "master" proxy server for that query) in the DHT overlay. The master caching module at that proxy server similarly checks its database cache and responds immediately if possible. Otherwise the master forwards the database query to the central database server, caches the query result, and returns the result to the original

proxy server which also caches the reply. Whenever a proxy server caches a database query result – whether for itself or as the master proxy server for that query – the consistency module first subscribes to some set of multicast groups related to that database query. (In Section 3 we describe how this set of groups is determined.) Whenever a database query result is removed from a proxy's cache the proxy server unsubscribes from any multicast groups that are no longer necessary for correct consistency maintenance.

When a dynamic web application issues a database update the proxy's local caching module always forwards the update directly to the centralized database server – which contains the persistent state of the system – and the proxy's consistency module publishes an update notification to some set of multicast groups related to that update. When a proxy server receives an update notification the consistency module invalidates any possibly-affected cached query results, removing them from the database cache.

Ferdinand's database query result cache is expected to be large and is implemented using disk-based storage. We chose our cache design for its simplicity rather than sophistication. A more complex cache design could drastically reduce cache size or potentially allow a proxy server to directly respond to queries not previously seen if it composed new results from the results of previous requests. A more complex cache design, however, would require more expensive cache processing. Our simple map design enables proxy servers to efficiently respond to database requests with minimal query processing.

Compared to non-cooperative proxy architectures, our cooperative caching design 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. Cooperative caching, however, increases the latency cost of an overall cache miss since the database query is first forwarded to its master proxy server before it is executed at the central database. If the latency between proxy servers is high, the cost of forwarding a database query to its master can be prohibitive, especially for web applications that execute many database queries for each web interaction. Cooperative database query caching also adds slight additional complexity to the cache consistency mechanism. In Section 5, this cost/benefit result is quantitatively explored through a series of experiments.

3. CACHE CONSISTENCY MANAGEMENT WITH GROUP-BASED MULTICAST

In Ferdinand a proxy subscribes to some set of multicast groups when it caches a database query and broadcasts to some set of multicast groups 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 affected database query must create a subscription to at least one multicast group 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.*

In this section we discuss the QUMA problem in more detail and show how to analyze a web application's embedded database requests to develop a correct, efficient QUMA solution for that application. We then describe our implementation of Ferdinand's consistency management in more detail and discuss the properties of the multicast system which are necessary to guarantee consistency maintenance.

3.1 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 group should affect each database query subscribed to that group. Good database query / update multicast associations also avoid creating subscriptions to groups to which updates will never be published and publications to groups for which no queries have caused subscriptions. An additional goal is to cluster related queries into the same multicast group, 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 multicast group 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 multicast group 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 this goal for a given web application using offline analysis of its database requests.

3.2 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 [15] 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 following example inventory application:

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 < ?
```

This example consists of two update templates and three database query templates, each with several parameters.

Template	Associated Multicast Groups
Template U1	{GroupU1:name=?, GroupU1}
Template U2	{GroupU2}
Template Q3	{GroupU1:name=?, GroupU2}
Template Q4	{GroupU1}
Template Q5	{GroupU1, GroupU2}

Figure 3: A correct QUMA solution for our sample inventory application.

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

Consider a multicast group based on the data affected by instantiations of Template U1, called GroupU1. Correct notification will occur for updates from this template if queries of Template Q3, Template Q4, and Template Q5 all create a subscription to GroupU1 and all such updates publish to GroupU1. 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 multicast group at runtime. In this association, queries of Template Q1 subscribe to GroupU1:name=? for the appropriate parameter binding. Updates published to the bound group GroupU1: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 multicast groups 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 multicast groups.

Figure 3 shows a correct QUMA solution for our sample inventory application. A question mark in the group 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 GroupU1:name=fork and GroupU2. If proxy server B then used Template U1 to insert a new item with the name "spoon" then B would publish update notifications to the groups GroupU1:name=spoon and GroupU1.

We previously described a wider range of QUMA solutions and created an evaluative framework to compare them as part of a technical report [12]. In practice, the solution we describe can efficiently map queries and updates to multicast groups 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 groups and the queries affected. Database requests using range or other selection predicate types, joins, or aggregates are typically mapped to multicast groups that do not embed run time parameters, resulting in a higher number of unnecessary update notifications.

3.3 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 only single-statement transactions are used. Ferdinand's consistency management system uses the offline workload analysis described above. We use a QUMA solution in which multicast groups correspond to the data clusters affected by each update template, much like the example QUMA solution. 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 multicast group there is a second "master" multicast group used for communication with master proxies. When a local cache miss occurs at a proxy server the proxy subscribes to all non-master groups 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 groups 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 groups 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 groups, to which any other affected proxy servers will be subscribed.

This hierarchical implementation with master and non-master groups is necessary to correctly maintain consistency because publish / subscribe does not constrain the order of notification delivery to different subscribers to the same group. If we did not use these master groups, 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.

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: (1) When a client subscribes to a publish / subscribe group, the subscription is confirmed to the client, and (2) A subscribed client is notified of all publications to a group that occur between the time the client's subscription to the group is confirmed and the time the client initiates an unsubscription from the group. Also, the central database must provide a slightly stronger guarantee than standard one-copy serializability: for non-concurrent transactions T_1 and T_2 at the central database such that T_1 commits before T_2 begins,

T_1 must appear before T_2 in the database's serial ordering of transactions. 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.*

For succinctness we only summarize the proof here and refer interested readers to Appendix A. In that section we precisely describe various algorithms at Ferdinand's proxy servers. We then use the above reliability guarantee 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 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.

4. FERDINAND PROTOTYPE IMPLEMENTATION

This section describes our implementation of the Ferdinand prototype. Ferdinand incorporates several open-source software projects and three components that we contribute: a main Ferdinand proxy module, a Ferdinand cache module, and a Ferdinand driver at the database server. All custom Ferdinand components are implemented with Java 1.5.0.

Each proxy server runs the Apache Tomcat server as a static cache and as a servlet container to execute web applications. The main Ferdinand module at each proxy is written as a Java 1.5.0 JDBC driver, that client applications use to interact with the database. The main proxy module communicates with the Ferdinand cache module, which transparently provides both local and cooperative caching and cache consistency management.

The Ferdinand cache module also executes at each proxy server. For cooperative caching we use the Pastry [24] overlay as our distributed hash table; Pastry is based on PRR trees [23] to route network messages to their destination. For consistency management we use the Scribe [6] publish / subscribe system. Scribe is a completely decentralized multicast system also based on Pastry. Scribe incurs no storage load for empty multicast groups and supports efficient subscription and unsubscription for large networks and large multicast groups, without introducing hot spots in the network. Scribe, however, does not guarantee reliable delivery of messages in cases of node failure. To accommodate this fact Ferdinand would flush all its caches in such an event, resulting in a temporary performance degradation but uninterrupted correct operation.

The Ferdinand cache module's disk-based cache map is implemented as a relation within a MySQL4 database management system. The relation consists of the the hash code of each cached query as well as serialized copies of the JDBC query and query result, and is indexed by the hash code. To retrieve a query result we first hash the JDBC query object, select all matching entries from the disk-based cache, and then select the desired entry from the result set if present.

The cache module communicates with the Ferdinand server module as needed to access the central database. The database server uses MySQL4 as the backend database, which our server module accesses locally using the MySQL Connector/J JDBC driver.

The evaluation of Ferdinand under various failure scenarios is future work. However, our use of a widely accepted DHT overlay and publish / subscribe implementation ensures that our experimental results faithfully reflect the overhead of failure management during normal operation.

5. EVALUATING THE PERFORMANCE OF FERDINAND

This section evaluates of our Ferdinand prototype. Ferdinand contains several enhancements over previous query caching work: distributed cooperative query caching, and the use of publish / subscribe as our consistency management infrastructure. Our primary goal was to determine the relative contribution of each enhancement to overall performance. To evaluate the performance contribution of our cooperative caching design we compared Ferdinand's performance to several alternative approaches commonly used to scale web applications. We then evaluated how the performance of DHT-based cooperative caching changes as inter-server network latency increases. Finally, we analyzed the contribution of publish / subscribe-based consistency management compared to a simpler broadcast-based system for propagating update notifications.

Section 5.1 describes the environment and benchmark web applications we used to evaluate Ferdinand. Section 5.2 discusses the performance of cooperative query caching and Section 5.3 describes our evaluation of Ferdinand's performance in higher-latency environments. Section 5.4 analyzes our publish / subscribe-based consistency management system.

5.1 Evaluation methodology

We conducted all experiments on the Emulab testbed [27]. Our central database server and each proxy server ran on 3 GHz Intel Pentium Xeon systems with 1 GB of memory and a large 10,000 RPM SCSI disk. Our experiments model two distinct network scenarios. First, we have a centralized proxy infrastructure in which the Ferdinand proxy network is co-located with the backend database in the same data center. For these experiments each proxy server was connected to the central database through a 100 Mbit switch. Benchmark clients were executed on separate 850 MHz client servers, directly connected to the proxy servers with high speed point-to-point links. Our second scenario models the Ferdinand infrastructure as being distributed by a third-party content delivery service. For this scenario we increased the proxy-to-proxy and proxy-to-database latencies, with the latency being an independent variable in the experiments.

We evaluated Ferdinand's performance using three benchmark applications: a modified version of the TPC-W bookstore [26], the RUBiS auction [21], and the RUBBoS bul-

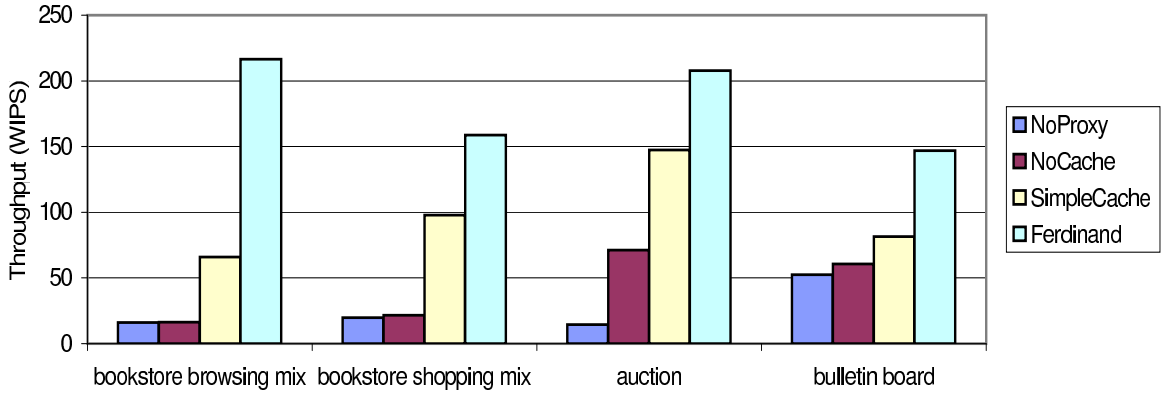

Figure 4: Throughput of Ferdinand compared to other scalability approaches.

letin board benchmarks. The TPC-W bookstore simulates the activity of users as they browse an online bookstore. Each emulated browser sends a request to the web server, waits for a reply, 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 each of our experiments we used the standard TPC-W think times and session times, exponentially distributed around means of 7 seconds and 15 minutes, respectively. We used a large configuration of the bookstore database composed of one million items and 86,400 registered users, with a total database size of 480 MB.

Our implementation of the bookstore benchmark contained one significant modification. In the original benchmark all books are uniformly popular. Our version used a more realistic Zipf distribution based on Brynjolfsson et al.'s work [5]. Specifically, we modeled 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.

The bookstore benchmark can be configured to generate distinct workloads with different distributions of queries and updates. We used two such configurations: the "browsing mix" and the "shopping mix." In the browsing mix 95% of emulated browser sessions do not make a purchase, while 5% do. In the shopping mix 20% of users make a purchase.

The RUBiS auction and RUBBoS bulletin board benchmarks also conform to the TPC-W model of emulated browsers, and we used similar browsing parameters of 7 seconds and 15 minutes. Our version of the RUBiS auction database contained 33,667 items and 100,000 registered users, totaling 990 MB. A typical web interaction in the auction benchmark requires more computation than for the bookstore, so even non-caching architectures can scale the benchmark well. The auction is extremely read-oriented compared to even the bookstore's browsing mix, but many auction queries embed the current time (e.g. "Show all auctions that end within an hour of now") and thus never result in a cache hit. Note that an application designer who was aware of a Ferdinand-like infrastructure might improve cache performance by rewriting these queries such that they were not all distinct. For the RUBBoS bulletin board benchmark we used a database containing 6,000 active stories, 60,000 old stories, 213,292 comments, and 500,000 users, totaling 1.6 GB. For the bulletin board some web interactions issue dozens of database requests, with each request being an efficient query whose result can be retrieved from an index (e.g. retrieving user information for every user who commented on a story). These web interactions often result in many cache hits but still require many interactions with the backend database.

Overall, the bookstore benchmark contains a higher proportion of update transactions than either the auction or bulletin board benchmarks. For the bookstore shopping mix about 14% of the database interactions are updates, while only 7% are updates for the auction and 2% for the bulletin board. These benchmarks are much more similar to each other, however, when considering the proportion of updates per web request rather than the proportion of updates per database request: 20-25% of web interactions cause updates for each benchmark. This difference is because the auction and bulletin board benchmarks sometimes execute many database queries for a single web request, as mentioned above.

For all benchmarks we partitioned user requests so that an assigned proxy server handled all requests from each particular user. All experiments were conducted with warm proxy caches, generated from long benchmark executions of about six hours. The primary metric for each benchmark is its maximum throughput produced for a given response-time threshold. For the TPC-W bookstore the maximum throughput is defined as the maximum number of web interactions per second (WIPS) sustained over a measurement period while responding to 90% of each type of interaction faster than a given response-time threshold for that interaction type. We used response-time thresholds for each interaction type as given in the TPC-W specification; for most servlets the interaction threshold is 3 seconds, but the threshold is longer for several interaction types. For the RUBiS auction and RUBBoS bulletin board benchmarks we defined the maximum throughput similarly, as the maximum WIPS sustained while keeping the response time under three seconds for 90% of all interactions. For the bookstore browsing mix we used a network of 12 proxy servers, and we used 8 proxy servers for the shopping mix, the auction benchmark, and the bulletin board benchmark. For all benchmarks we generated the necessary indices and configured the backend database to attempt to maximize performance

	SimpleCache	Ferdinand
bookstore browsing mix	17%	7%
bookstore shopping mix	22%	14%
auction	40%	17%
bulletin board	20%	11%

Table 1: Cache miss rates for Ferdinand and SimpleCache.

for a centralized environment – including enabling MySQL's internal query result caching feature – so that Ferdinand's performance gains are not merely a reflection of inefficiencies at the central database server.

5.2 The performance of DHT-based cooperative query caching

To evaluate the usefulness of cooperative caching we compared performance of four approaches: (1) NoProxy, a centralized three-tier system with no proxy servers and only the centralized database server cache, (2) NoCache, a centralized database server with proxies that executed the web and application server but performed no proxy caching, (3) SimpleCache, a Ferdinand-like system where each proxy server had a stand-alone query cache but forwarded all cache misses directly to the central database, and (4) Ferdinand, which extends SimpleCache with a cooperative proxy cache. NoProxy included Apache Tomcat and MySQL at the central server much like a Ferdinand proxy. Like Ferdinand, SimpleCache also used a disk-based query cache and publish / subscribe for cache consistency management.

Figure 4 shows the maximum throughput, under the response-time threshold, for each caching approach and each of our benchmarks, and Table 1 shows the average cache miss rates for SimpleCache and Ferdinand. 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 performance 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 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%

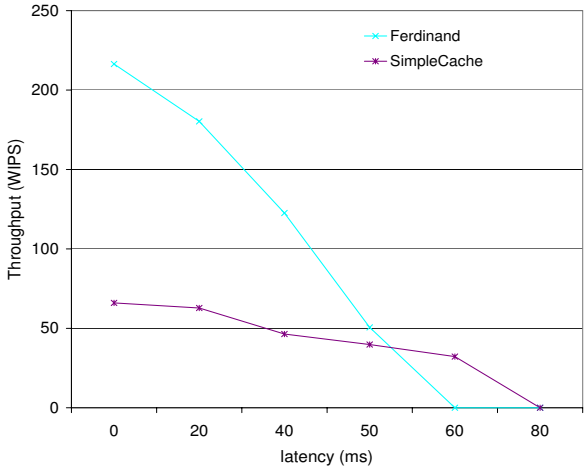

Figure 5: Throughputs of Ferdinand and SimpleCache as a function of round trip server-to-server latency.

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 cache hit 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.

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.

5.3 Ferdinand in higher-latency environments

Our next experiment evaluates 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. 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 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 [17]. 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.

5.4 The performance of publish / subscribe consistency management

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

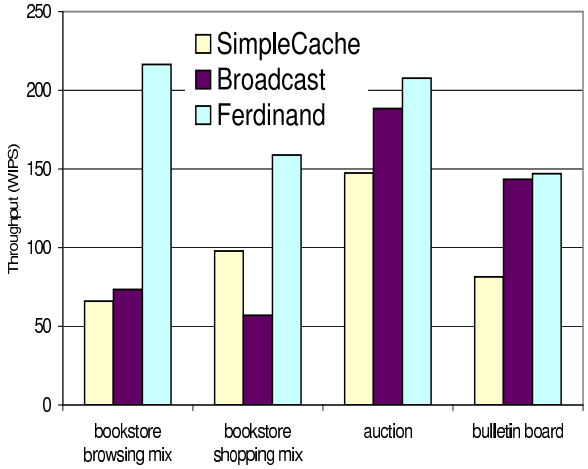

Figure 6: Throughput of Ferdinand compared to broadcast-based consistency management.

Figure 6 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.

Overall these results demonstrate 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. RELATED WORK

There exists a large body of work on scaling the database component in three-tiered architectures. IBM DBCache [2, 17], IBM DBProxy [3], and NEC CachePortal [16] all implement database query caching for web applications. Of these DBProxy uses a relation-based cache representation and sophisticated query processing at each proxy server, en-

abling a proxy to generate a response to an uncached query if the result can be composed from data retrieved by other cached queries. Each of these systems uses some form of centralized consistency management and implements the same consistency model as Ferdinand: each cache is kept coherent with the central database, but transactional consistency is not guaranteed for multi-statement transactions that use the query cache.

An alternative approach to query caching is to explicitly replicate the database at each proxy server. Full database replication, such as that done by C-JDBC [20], requires all updates to be broadcast to every proxy server and does not scale well on many workloads. Partial replication of the database at each proxy server can reduce the cost of consistency management, but also reduces the set of queries that each proxy can answer and raises the question of what data to place at each proxy server. Some promising work in this area includes GlobeDB [25] and GlobeTP [13], both of which focus on automatic data placement in partial replication systems. In particular, GlobeTP uses a template-based analysis similar to the analysis we use to generate a query / update multicast association. Like the query caching systems above, both GlobeDB and GlobeTP implement a consistency model similar Ferdinand's. One key advantage of Ferdinand (and other query caches) over these systems is its ability to automatically adapt to changes in the workload over time without having to re-partition or re-replicate the database.

Distributed hash tables have been used to implement caches in several other settings, including a number of peer-to-peer caches of static web content, e.g. Squirrel [14]. To the best of our knowledge no one has previously studied the performance of DHTs for database query caching; without scalable consistency management, distributed caches offer little advantage over traditional caching systems.

Group communication has long been among the tools used to facilitate database replication, e.g. Gustavo Alonso's 1997 work [1]. Recently, Chandramouli et al. [10] used content-based publish / subscribe to propagate update notifications for distributed continuous queries, much as we do using topic-based publish / subscribe for consistency management. In addition to the difference in application setting, a key distinction between their work and Ferdinand is our use of offline analysis to improve the mapping of database requests into the publish / subscribe system.

Others have used application-level analysis to improve the performance of replication. Amiri et al. [4] exploited templates to scale consistency maintenance, while Gao et al. [11] demonstrate application-specific optimizations to improve replication performance for TPC-W. Challenger et al. [8] analyzed the data dependencies between dynamic web pages and the back-end database to maintain the coherence of the dynamic content in proxy caches. Challenger et al. [9] later extended their work to determining the dependencies between fragments of dynamic web pages and the backend database, and in [7] Chabbouh and Makpangou showed how to use an offline analysis similar to ours to determine the dependencies between a web application's templated database requests and the dynamic page fragments they affect.

Finally, we have previously studied several problems related to database query result caches in CDN-like environments. In [18] we considered the problem of maintaining the consistency of private data in such a system. We showed that the cost of consistency maintenance often increased as more stringent privacy was required, and developed analytical techniques to predict the scalability of consistency maintenance given a particular set of privacy requirements. In [19] we showed how to automatically modify queries and updates to improve the accuracy of invalidation of private data in the setting above. This paper extends our previous work in several key ways. First, all of our previous work used a single-tier cache design, in which each proxy server cached results alone without sharing results with other proxy servers. All of our previous experimental work only evaluated the feasibility of query result caching by measuring the performance of single-proxy implementations: this paper represents our first full-scale implementation and evaluation of a proxy-based caching system. Finally, although we outlined our high-level design for publish / subscribe consistency management in [22], this paper is the first evaluation of our full architecture and is the first quantitative demonstration that simple cache consistency management can be the scalability bottleneck even when all updates are sent to a central back-end database server.

7. 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 paper introduces 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. A key challenge in using publish / subscribe for consistency management is to map database queries and updates to multicast groups to ensure that an update's notification is sent to every cache affected by that update. In this paper we showed how to use an offline analysis of queries and updates to determine their dependencies and create an efficient database query / update multicast association.

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. Finally, we demonstrated that both of our major enhancements – improved cache hit rates and efficient consistency management – are important factors in Ferdinand's overall performance gain.

7.1 Future work

We continue to work on several questions introduced by Ferdinand and other database query-caching systems. If Ferdinand is implemented in a CDN-like environment and

latency is introduced between the database and application servers, performance is significantly degraded for applications that require repeated interactions with the database. Existing applications, however, are not written with consideration of this environment and can be needlessly inefficient in that setting. We are developing automated compiler optimization-like techniques to improve the application-database interaction for such an environment.

Also, our implementation of Ferdinand attempts to cache every database query result regardless of the likelihood that the result will be used before it is invalidated. Excessive caching of unused database query results can tax the consistency management system since each such occurrence might require a subscription, notification of an invalidating update, and unsubscription from multiple multicast groups. We seek automated offline and adaptive online techniques to help prevent the caching of database query results unlikely to be used.

Finally, we seek to better evaluate Ferdinand against competing methods. We are developing an efficient directory-based consistency management implementation to further evaluate the contribution of publish / subscribe.

8. REFERENCES

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

[2] M. Altinel, C. Bornhovd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proc. International Conference on Very Large Data Bases*, 2003.

[3] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proc. International Conference on Data Engineering*, 2003.

[4] K. Amiri, S. Sprenkle, R. Tewari, and S. Padmanabhan. Exploiting templates to scale consistency maintenance in edge database caches. In *Proc. International Workshop on Web Content Caching and Distribution*, 2003.

[5] E. Brynojolfsson, M. Smith, and Y. Hu. Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. MIT Sloan Working paper No. 4305-03, 2003.

[6] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, October 2002.

[7] I. Chabbouh and M. Makpangou. Caching dynamic content with automatic fragmentation. In G. Kotsis, D. Taniar, S. Bressan, I. K. Ibrahim, and S. Mokhtar, editors, *iiWAS*, volume 196 of *books@ocg.at*, pages 975–986. Austrian Computer Society, 2005.

[8] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, 1999.

[9] J. Challenger, P. Dantzig, A. Iyengar, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Trans. Internet Techn.*, 5(2):359–389, 2005.

[10] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2006.

[11] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Improving availability and performance with application-specific data replication. *IEEE Transactions on Knowlege and Data Engineering*, 17(1):106–120, Jan 2005.

[12] C. Garrod, A. Manjhi, A. Ailamaki, P. Gibbons, B. M. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable consistency management for web database caches. Technical Report CMU-CS-06-128, Carnegie Mellon University, July 2006.

[13] T. Groothuyse, S. Sivasubramanian, and G. 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, *Proc. International Conference on the World Wide Web*, pages 301–310. ACM, 2007.

[14] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. 21st ACM SIGACT-SIGOPS Principles of Distributed Computing*, 2002.

[15] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *Proc. International Conference on Very Large Data Bases*, 1993.

[16] W. Li, O. Po, W. Hsiung, K. S. Candan, D Agrawal, Y. Akca, and K Taniguchi. CachePortal II: Acceleration of very large scale data center-hosted database-driven web applications. In *Proc. International Conference on Very Large Data Bases*, 2003.

[17] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2002.

[18] A. Manjhi, A. Ailamaki, B. M. Maggs, T. C. Mowry, C. Olston, and A. Tomasic. Simultaneous scalability and security for data-intensive web applications. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2006.

[19] A. Manjhi, P. B. Gibbons, A. Ailamaki, C. Garrod, B. M. Maggs, T. C. Mowry, C. Olston, A. Tomasic, and H. Yu. Invalidation clues for database scalability services. In *Proc. International Conference on Data Engineering*, 2007.

[20] ObjectWeb Consortium. C-JDBC: Flexible database clustering middleware. `http://c-jdbc.objectweb.org/`.

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

[22] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. Maggs, and T. Mowry. A scalability service for dynamic web applications. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2005.

[23] C. G. Plaxton, R. Rajaraman, and A. W. Richa.

Accessing nearby copies of replicated objects in a distributed environment. Theory of Computing Systems, 32:241-280, 1999.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001.

[25] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *Proc. International Conference on the World Wide Web*, 2005.

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

[27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. Symposium on Operating Systems Design and Implementation*, 2002.

APPENDIX

A. CORRECTNESS OF FERDINAND'S CONSISTENCY MANAGEMENT

In this section we prove that Ferdinand's consistency management mechanism correctly invalidates all affected cached query results at each Ferdinand proxy server. We first describe the relevant Ferdinand algorithms: how queries, updates, and update notifications are processed by the proxy servers. We then prove our central theorem, first by showing that each affected query result is purged from the cache at that result's master server and then showing that it is purged from all other proxy caches.

The algorithm for processing a query at a local non-master proxy is:

```
LQ1.  Check cache for Q, return query result if
      present
LQ2.  Subscribe to all non-master groups related to
      Q, wait for confirmation
LQ3.  Hash Q and forward to master proxy for Q,
      wait for reply
LQ4   Apply any notifications received since (2),
      add result to cache if it was not invalidated
LQ5.  Return query result
```

The algorithm for processing a query at its master proxy is:

```
MQ1.  Check cache for Q, return query result if
      present
MQ2.  Subscribe to all master groups related to Q,
      wait for confirmation
MQ3.  Forward Q to central database server, wait
      for reply
MQ4.  Apply any notifications received since (2),
      add result to cache if it was not invalidated
MQ5.  Return query result
```

The algorithm for processing an update:

```
U1.   Forward update to central database server,
      wait for reply
U2.   Publish update to all related master groups
U3.   Return update result
```

The algorithm for processing a master-group notification:

```
MN1.  Apply update, removing any affected queries
      from cache and marking any pending queries as
      invalid
MN2.  Republish update to all related non-master
      groups
MN3.  Unsubscribe from any master groups on which
      no remaining or pending queries depend
```

The algorithm for processing a non-master notification:

```
LN1.  Apply update, removing any affected queries
      from cache and marking any pending queries as
      invalid
LN2.  Unsubscribe from any groups on which no
      remaining cached or pending queries depend
```

We assume that the network, publish / subscribe system and database are reliable, and that the database's serializability guarantee is slightly stronger than standard one-copy serializability. In particular:

A1 The publish / subscribe system confirms subscriptions.

A2 A subscribed client will be notified of all publications to a group that occur between the time the client's subscription is confirmed and the time the client initiates an unsubscription from the group.

A3 If transactions T_1 and T_2 are non-concurrent at the central database such that T_1 commits before T_2 begins, T_1 appears before T_2 in the database's serial ordering of transactions.

These assumptions allow us to prove our central 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 in two parts. We first show that 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` (step MQ2) and let t_{pub} be the time at which notification of U is published to `G` (step

U2), 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` occured 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 $\mathcal{C}$ be any non-master proxy for Q that is already caching or will ever receive query result q. Then $\mathcal{C}$ 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 $\mathcal{C}$ has confirmed its subscription to `G` (step LQ2) and let t_{pub} be the time at which Q's master publishes its invalidation to `G` (step MN2). 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 (step MN2). 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 $\mathcal{C}$ has received no invalidations for q by time t_{pub}. By the same logic we have that $\mathcal{C}$ will receive a notification of U' and invalidate q. □

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