Consistency-preserving Caching of Dynamic Database Content*

Niraj Tolia and M. Satyanarayanan
Carnegie Mellon University
{ntolia,satya}@cs.cmu.edu

ABSTRACT

With the growing use of dynamic web content generated from relational databases, traditional caching solutions for throughput and latency improvements are ineffective. We describe a middleware layer called *Ganesh* that reduces the volume of data transmitted without semantic interpretation of queries or results. It achieves this reduction through the use of cryptographic hashing to detect similarities with previous results. These benefits do not require any compromise of the strict consistency semantics provided by the back-end database. Further, Ganesh does not require modifications to applications, web servers, or database servers, and works with closed-source applications and databases. Using two benchmarks representative of dynamic web sites, measurements of our prototype show that it can increase end-to-end throughput by as much as twofold for non-data intensive applications and by as much as tenfold for data intensive ones.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; H.2.4 [**Database Management**]: Systems

General Terms

Design, Performance

Keywords

content addressable storage, relational database systems, database caching, wide area networks, bandwidth optimization

1. INTRODUCTION

An increasing fraction of web content is dynamically generated from back-end relational databases. Even when database content remains unchanged, temporal locality of access cannot be exploited because dynamic content is not cacheable by web browsers or by intermediate caching servers such as Akamai mirrors. In a multi-tiered architecture, each web request can stress the WAN link between the web server and the database. This causes user experience to be highly variable because there is no caching to insulate the client from bursty loads. Previous attempts in caching dynamic database content have generally weakened transactional semantics [3, 4] or required application modifications [15, 34].

We report on a new solution that takes the form of a database-agnostic middleware layer called *Ganesh*. Ganesh makes no effort to semantically interpret the contents of queries or their results. Instead, it relies exclusively on cryptographic hashing to detect similarities with previous results. Hash-based similarity detection has seen increasing use in distributed file systems [26, 36, 37] for improving performance on low-bandwidth networks. However, these techniques have not been used for relational databases. Unlike previous approaches that use generic methods to detect similarity, Ganesh exploits the structure of relational database results to yield superior performance improvement.

One faces at least three challenges in applying hash-based similarity detection to back-end databases. First, previous work in this space has traditionally viewed storage content as uninterpreted bags of bits with no internal structure. This allows hash-based techniques to operate on long, contiguous runs of data for maximum effectiveness. In contrast, relational databases have rich internal structure that may not be as amenable to hash-based similarity detection. Second, relational databases have very tight integrity and consistency constraints that must not be compromised by the use of hash-based techniques. Third, the source code of commercial databases is typically not available. This is in contrast to previous work which presumed availability of source code.

Our experiments show that Ganesh, while conceptually simple, can improve performance significantly at bandwidths representative of today's commercial Internet. On benchmarks modeling multi-tiered web applications, the throughput improvement was as high as tenfold for data-intensive workloads. For workloads that were not data-intensive, throughput improvements of up to twofold were observed. Even when bandwidth was not a constraint, Ganesh had low overhead and did not hurt performance. Our experiments also confirm that exploiting the structure present in database results is crucial to this performance improvement.

2. BACKGROUND

2.1 Dynamic Content Generation

As the World Wide Web has grown, many web sites have decentralized their data and functionality by pushing them to the edges of the Internet. Today, eBusiness systems often use a three-tiered architecture consisting of a front-end web server, an application server, and a back-end database server. Figure 1 illustrates this architecture. The first two tiers can be replicated close to a concentration of clients at the edge of the Internet. This improves user experience by lowering end-to-end latency and reducing exposure

*This research was supported by the National Science Foundation (NSF) under grant number CCR-0205266. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or Carnegie Mellon University.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

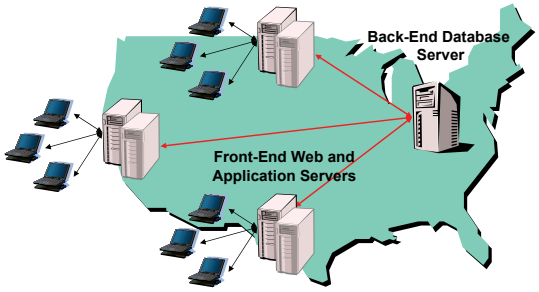


Figure 1: Multi-Tier Architecture

to backbone traffic congestion. It can also increase the availability and scalability of web services.

Content that is generated dynamically from the back-end database cannot be cached in the first two tiers. While databases can be easily replicated in a LAN, this is infeasible in a WAN because of the difficult task of simultaneously providing strong consistency, availability, and tolerance to network partitions [7]. As a result, databases tend to be centralized to meet the strong consistency requirements of many eBusiness applications such as banking, finance, and online retailing [38]. Thus, the back-end database is usually located far from many sets of first and second-tier nodes [2]. In the absence of both caching and replication, WAN bandwidth can easily become a limiting factor in the performance and scalability of data-intensive applications.

2.2 Hash-Based Systems

Ganesh's focus is on efficient transmission of results by discovering similarities with the results of previous queries. As SQL queries can generate large results, hash-based techniques lend themselves well to the problem of efficiently transferring these large results across bandwidth constrained links.

The use of hash-based techniques to reduce the volume of data transmitted has emerged as a common theme of many recent storage systems, as discussed in Section 8.2. These techniques rely on some basic assumptions. Cryptographic hash functions are assumed to be collision-resistant. In other words, it is computationally intractable to find two inputs that hash to the same output. The functions are also assumed to be one-way; that is, finding an input that results in a specific output is computationally infeasible. Menezes et al. [23] provide more details about these assumptions.

The above assumptions allow hash-based systems to assume that collisions do not occur. Hence, they are able to treat the hash of a data item as its unique identifier. A collection of data items effectively becomes content-addressable, allowing a small hash to serve as a codeword for a much larger data item in permanent storage or network transmission.

The assumption that collisions are so rare as to be effectively non-existent has recently come under fire [17]. However, as explained by Black [5], we believe that these issues do not form a concern for Ganesh. All communication is between trusted parts of the system and an adversary has no way to force Ganesh to accept invalid data. Further, Ganesh does not depend critically on any specific hash function. While we currently use SHA-1, replacing it with a different hash function would be simple. There would be no impact on performance as stronger hash functions (e.g. SHA-256) only add a few extra bytes and the generated hashes are still orders of magnitude smaller than the data items they represent. No re-hashing of permanent storage is required since Ganesh only uses hashing on volatile data.

3. DESIGN AND IMPLEMENTATION

Ganesh exploits redundancy in the result stream to avoid transmitting result fragments that are already present at the query site. Redundancy can arise naturally in many different ways. For example, a query repeated after a certain interval may return a different result because of updates to the database; however, there may be significant commonality between the two results. As another example, a user who is refining a search may generate a sequence of queries with overlapping results. When Ganesh detects redundancy, it suppresses transmission of the corresponding result fragments. Instead, it transmits a much smaller digest of those fragments and lets the query site reconstruct the result through hash lookup in a cache of previous results. In effect, Ganesh uses computation at the edges to reduce Internet communication.

Our description of Ganesh focuses on four aspects. We first explain our approach to detecting similarity in query results. Next, we discuss how the Ganesh architecture is completely invisible to all components of a multi-tier system. We then describe Ganesh's proxy-based approach and the dataflow for detecting similarity.

3.1 Detecting Similarity

One of the key design decisions in Ganesh is how similarity is detected. There are many potential ways to decompose a result into fragments. The optimal way is, of course, the one that results in the smallest possible object for transmission for a given query's results. Finding this optimal decomposition is a difficult problem because of the large space of possibilities and because the optimal choice depends on many factors such as the contents of the query's result, the history of recent results, and the cache management algorithm.

When an object is opaque, the use of Rabin fingerprints [8, 30] to detect common data between two objects has been successfully shown in the past by systems such as LBFS [26] and CASPER [37]. Rabin fingerprinting uses a sliding window over the data to compute a rolling hash. Assuming that the hash function is uniformly distributed, a chunk boundary is defined whenever the lower order bits of the hash value equal some predetermined value. The number of lower order bits used defines the average chunk size. These sub-divided chunks of the object become the unit of comparison for detecting similarity between different objects.

As the locations of boundaries found by using Rabin fingerprints is stochastically determined, they usually fail to align with any structural properties of the underlying data. The algorithm therefore deals well with *in-place* updates, insertions and deletions. However, it performs poorly in the presence of any reordering of data.

Figure 2 shows an example where two results, `A` and `B`, consisting of three rows, have the same data but have different sort attributes. In the extreme case, Rabin fingerprinting might be unable to find any similar data due to the way it detects chunk boundaries. Fortunately, Ganesh can use domain specific knowledge for more precise boundary detection. The information we exploit is that a query's result reflects the structure of a relational database where all data is organized as tables and rows. It is therefore simple to check for similarity with previous results at two granularities: first the entire result, and then individual rows. The end of a row in a result serves as a natural chunk boundary. It is important to note that using the tabular structure in results only involves shallow interpretation of the data. Ganesh does not perform any deeper semantic interpretation such as understanding data types, result schema, or integrity constraints.

Tuning Rabin fingerprinting for a workload can also be difficult. If the average chunk size is too large, chunks can span multiple result rows. However, selecting a smaller average chunk size increases the amount of metadata required to the describe the results.

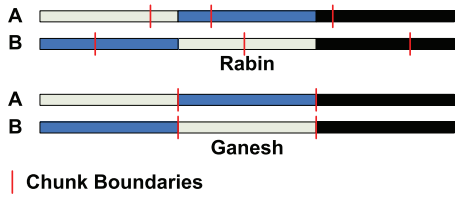


Figure 2: Rabin Fingerprinting vs. Ganesh's Chunking

This, in turn, would decrease the savings obtained via its use. Rabin fingerprinting also needs two computationally-expensive passes over the data: once to determine chunk boundaries and one again to generate cryptographic hashes for the chunks. Ganesh only needs a single pass for hash generation as the chunk boundaries are provided by the data's natural structure.

The performance comparison in Section 6 shows that Ganesh's row-based algorithm outperforms Rabin fingerprinting. Given that previous work has already shown that Rabin fingerprinting performs better than gzip [26], we do not compare Ganesh to compression algorithms in this paper.

3.2 Transparency

The key factor influencing our design was the need for Ganesh to be completely transparent to all components of a typical eBusiness system: web servers, application servers, and database servers. Without this, Ganesh stands little chance of having a significant real-world impact. Requiring modifications to any of the above components would raise the barrier for entry of Ganesh into an existing system, and thus reduce its chances of adoption. Preserving transparency is simplified by the fact that Ganesh is purely a performance enhancement, not a functionality or usability enhancement.

We chose *agent interposition* as the architectural approach to realizing our goal. This approach relies on the existence of a compact programming interface that is already widely used by target software. It also relies on a mechanism to easily add new code without disrupting existing module structure.

These conditions are easily met in our context because of the popularity of Java as the programming language for eBusiness systems. The Java Database Connectivity (JDBC) API [32] allows Java applications to access a wide variety of databases and even other tabular data repositories such as flat files. Access to these data sources is provided by JDBC drivers that translate between the JDBC API and the database communication mechanism. Figure 3(a) shows how JDBC is typically used in an application.

As the JDBC interface is standardized, one can substitute one JDBC driver for another without application modifications. The JDBC driver thus becomes the natural module to exploit for code interposition. As shown in Figure 3(b), the native JDBC driver is replaced with a Ganesh JDBC driver that presents the same standardized interface. The Ganesh driver maintains an in-memory cache of result fragments from previous queries and performs reassembly of results. At the database, we add a new process called the Ganesh proxy. This proxy, which can be shared by multiple front-end nodes, consists of two parts: code to detect similarity in result fragments and the original native JDBC driver that communicates with the database. The use of a proxy at the database makes Ganesh database-agnostic and simplifies prototyping and experimentation. Ganesh is thus able to work with a wide range of databases and applications, requiring no modifications to either.

3.3 Proxy-Based Caching

The native JDBC driver shown in Figure 3(a) is a lightweight code component supplied by the database vendor. Its main function is to mediate communication between the application and the remote database. It forwards queries, buffers entire results, and responds to application requests to view parts of results.

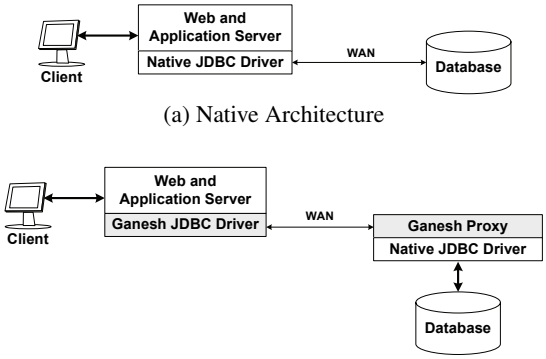


Figure 3: Native vs. Ganesh Architecture

The Ganesh JDBC driver shown in Figure 3(b) presents the application with an interface identical to that provided by the native driver. It provides the ability to reconstruct results from compact hash-based descriptions sent by the proxy. To perform this reconstruction, the driver maintains an in-memory cache of recently-received results. This cache is only used as a source of result fragments in reconstructing results. No attempt is made by the Ganesh driver or proxy to track database updates. The lack of cache consistency does not hurt correctness as a description of the results is always fetched from the proxy — at worst, there will be no performance benefit from using Ganesh. Stale data will simply be paged out of the cache over time.

The Ganesh proxy accesses the database via the native JDBC driver, which remains unchanged between Figures 3(a) and (b). The database is thus completely unaware of the existence of the proxy. The proxy does not examine any queries received from the Ganesh driver but passes them to the native driver. Instead, the proxy is responsible for inspecting database output received from the native driver, detecting similar results, and generating hash-based encodings of these results whenever enough similarity is found. While this architecture does not decrease the load on a database, as mentioned earlier in Section 2.1, it is much easier to replicate databases for scalability in a LAN than in a WAN.

To generate a hash-based encoding, the proxy must be aware of what result fragments are available in the Ganesh driver's cache. One approach is to be optimistic, and to assume that all result fragments are available. This will result in the smallest possible initial transmission of a result. However, in cases where there is little overlap with previous results, the Ganesh driver will have to make many calls to the proxy during reconstruction to fetch missing result fragments. To avoid this situation, the proxy loosely tracks the state of the Ganesh driver's cache. Since both components are under our control, it is relatively simple to do this without resorting to gray-box techniques or explicit communication for maintaining cache coherence. Instead, the proxy simulates the Ganesh driver's cache management algorithm and uses this to maintain a list of hashes for which the Ganesh driver is likely to possess the result fragments. In case of mistracking, there will be no loss of correctness but there will be extra round-trip delays to fetch the missing fragments. If the client detects loss of synchronization with the proxy, it can ask the proxy to reset the state shared between them. Also note that the proxy does not need to keep the result fragments themselves, only their hashes. This allows the proxy to remain scalable even when it is shared by many front-end nodes.

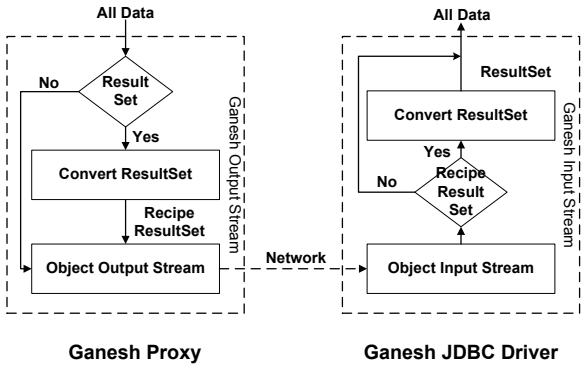


Figure 4: Dataflow for Result Handling

3.4 Encoding and Decoding Results

The Ganesh proxy receives database output as Java objects from the native JDBC driver. It examines this output to see if a Java object of type `ResultSet` is present. The JDBC interface uses this data type to store results of database queries. If a `ResultSet` object is found, it is shrunk as discussed below. All other Java objects are passed through unmodified.

As discussed in Section 3.1, the proxy uses the row boundaries defined in the `ResultSet` to partition it into fragments consisting of single result rows. All `ResultSet` objects are converted into objects of a new type called `RecipeResultSet`. We use the term "recipe" for this compact description of a database result because of its similarity to a *file recipe* in the CASPER file system [37]. The conversion replaces each result fragment that is likely to be present in the Ganesh driver's cache by a SHA-1 hash of that fragment. Previously unseen result fragments are retained verbatim. The proxy also retains hashes for the new result fragments as they will be present in the driver's cache in the future. Note that the proxy only caches hashes for result fragments and does not cache recipes.

The proxy constructs a `RecipeResultSet` by checking for similarity at the entire result and then the row level. If the entire result is predicted to be present in the Ganesh driver's cache, the `RecipeResultSet` is simply a single hash of the entire result. Otherwise, it contains hashes for those rows predicted to be present in that cache; all other rows are retained verbatim. If the proxy estimates an overall space savings, it will transmit the `RecipeResultSet`. Otherwise the original `ResultSet` is transmitted.

The `RecipeResultSet` objects are transformed back into `ResultSet` objects by the Ganesh driver. Figure 4 illustrates `ResultSet` handling at both ends. Each SHA-1 hash found in a `RecipeResultSet` is looked up in the local cache of result fragments. On a hit, the hash is replaced by the corresponding fragment. On a miss, the driver contacts the Ganesh proxy to fetch the fragment. All previously unseen result fragments that were retained verbatim by the proxy are hashed and added to the result cache.

There should be very few misses if the proxy has accurately tracked the Ganesh driver's cache state. A future optimization would be to batch the fetch of missing fragments. This would be valuable when there are many small missing fragments in a high-latency WAN. Once the transformation is complete, the fully reconstructed `ResultSet` object is passed up to the application.

4. EXPERIMENTAL VALIDATION

Three questions follow from the goals and design of Ganesh:

- First, can performance can be improved significantly by exploiting similarity across database results?
- Second, how important is Ganesh's structural similarity detection relative to Rabin fingerprinting's similarity detection?
- Third, is the overhead of the proxy-based design acceptable?

Benchmark	Dataset	Details
`BBOARD`	2.0 GB	500,000 Users, 12,000 Stories 3,298,000 Comments
`AUCTION`	1.3 GB	1,000,000 Users, 34,000 Items

Table 1: Benchmark Dataset Details

Our evaluation answers these question through controlled experiments with the Ganesh prototype. This section describes the benchmarks used, our evaluation procedure, and the experimental setup. Results of the experiments are presented in Sections 5, 6, and 7.

4.1 Benchmarks

Our evaluation is based on two benchmarks [18] that have been widely used by other researchers to evaluate various aspects of multi-tier [27] and eBusiness architectures [9]. The first benchmark, `BBOARD`, is modeled after *Slashdot*, a technology-oriented news site. The second benchmark, `AUCTION`, is modeled after *eBay*, an online auction site. In both benchmarks, most content is dynamically generated from information stored in a database. Details of the datasets used can be found in Table 1.

4.1.1 *The* `BBOARD` *Benchmark*

The `BBOARD` benchmark, also known as RUBBoS [18], models Slashdot, a popular technology-oriented web site. Slashdot aggregates links to news stories and other topics of interest found elsewhere on the web. The site also serves as a bulletin board by allowing users to comment on the posted stories in a threaded conversation form. It is not uncommon for a story to gather hundreds of comments in a matter of hours. The `BBOARD` benchmark is similar to the site and models the activities of a user, including read-only operations such as browsing the stories of the day, browsing story categories, and viewing comments as well as write operations such as new user registration, adding and moderating comments, and story submission.

The benchmark consists of three different phases: a short warm-up phase, a runtime phase representing the main body of the workload, and a short cool-down phase. In this paper we only report results from the runtime phase. The warm-up phase is important in establishing dynamic system state, but measurements from that phase are not significant for our evaluation. The cool-down phase is solely for allowing the benchmark to shut down.

The warm-up, runtime, and cool-down phases are 2, 15, and 2 minutes respectively. The number of simulated clients were 400, 800, 1200, and 1600. The benchmark is available in a Java Servlets and PHP version and has different datasets; we evaluated Ganesh using the Java Servlets version and the Expanded dataset.

The `BBOARD` benchmark defines two different workloads. The first, the *Authoring* mix, consists of 70% read-only operations and 30% read-write operations. The second, the *Browsing* mix, contains only read-only operations and does not update the database.

4.1.2 *The* `AUCTION` *Benchmark*

The `AUCTION` benchmark, also known as RUBiS [18], models eBay, the online auction site. The eBay web site is used to buy and sell items via an auction format. The main activities of a user include browsing, selling, or bidding for items. Modeling the activities on this site, this benchmark includes read-only activities such as browsing items by category and by region, as well as read-write

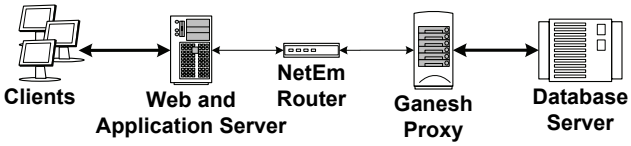


Figure 5: Experimental Setup

activities such as bidding for items, buying and selling items, and leaving feedback.

As with BBOARD, the benchmark consists of three different phases. The warm-up, runtime, and cool-down phases for this experiment are 1.5, 15, and 1 minutes respectively. We tested Ganesh with four client configurations where the number of test clients was set to 400, 800, 1200, and 1600. The benchmark is available in a Enterprise Java Bean (EJB), Java Servlets, and PHP version and has different datasets; we evaluated Ganesh with the Java Servlets version and the Expanded dataset.

The AUCTION benchmark defines two different workloads. The first, the *Bidding* mix, consists of 70% read-only operations and 30% read-write operations. The second, the *Browsing* mix, contains only read-only operations and does not update the database.

4.2 Experimental Procedure

Both benchmarks involve a synthetic workload of clients accessing a web server. The number of clients emulated is an experimental parameter. Each emulated client runs an instance of the benchmark in its own thread, using a matrix to transition between different benchmark states. The matrix defines a stochastic model with probabilities of transitioning between the different states that represent typical user actions. An example transition is a user logging into the AUCTION system and then deciding on whether to post an item for sale or bid on active auctions. Each client also models user think time between requests. The think time is modeled as an exponential distribution with a mean of 7 seconds.

We evaluate Ganesh along two axes: number of clients and WAN bandwidth. Higher loads are especially useful in understanding Ganesh's performance when the CPU or disk of the database server or proxy is the limiting factor. A previous study has shown that approximately 50% of the wide-area Internet bottlenecks observed had an available bandwidth under 10 Mb/s [1]. Based on this work, we focus our evaluation on the WAN bandwidth of 5 Mb/s with 66 ms of round-trip latency, representative of severely constrained network paths, and 20 Mb/s with 33 ms of round-trip latency, representative of a moderately constrained network path. We also report Ganesh's performance at 100 Mb/s with no added round-trip latency. This bandwidth, representative of an unconstrained network, is especially useful in revealing any potential overhead of Ganesh in situations where WAN bandwidth is not the limiting factor. For each combination of number of clients and WAN bandwidth, we measured results from the two configurations listed below:

- *Native:* This configuration corresponds to Figure 3(a). Native avoids Ganesh's overhead in using a proxy and performing Java object serialization.
- *Ganesh:* This configuration corresponds to Figure 3(b). For a given number of clients and WAN bandwidth, comparing these results to the corresponding *Native* results gives the performance benefit due to the Ganesh middleware system.

The metric used to quantify the improvement in throughput is the number of client requests that can be serviced per second. The metric used to quantify Ganesh's overhead is the average response time for a client request. For all of the experiments, the Ganesh driver used by the application server used a cache size of 100,000 items[1]. The proxy was effective in tracking the Ganesh driver's cache state; for all of our experiments the miss rate on the driver never exceeded 0.7%.

4.3 Experimental Setup

The experimental setup used for the benchmarks can be seen in Figure 5. All machines were 3.2 GHz Pentium 4s (with Hyper-Threading enabled.) With the exception of the database server, all machines had 2 GB of SDRAM and ran the Fedora Core Linux distribution. The database server had 4 GB of SDRAM.

We used Apache's Tomcat as both the application server that hosted the Java Servlets and the web server. Both benchmarks used Java Servlets to generate the dynamic content. The database server used the open source MySQL database. For the native JDBC drivers, we used the Connector/J drivers provided by MySQL. The application server used Sun's Java Virtual Machine as the runtime environment for the Java Servlets. The sysstat tool was used to monitor the CPU, network, disk, and memory utilization on all machines.

The machines were connected by a switched gigabit Ethernet network. As shown in Figure 5, the front-end web and application server was separated from the proxy and database server by a NetEm router [16]. This router allowed us to control the bandwidth and latency settings on the network. The NetEm router is a standard PC with two network cards running the Linux Traffic Control and Network Emulation software. The bandwidth and latency constraints were only applied to the link between the application server and the database for the native case and between the application server and the proxy for the Ganesh case. There is no communication between the application server and the database with Ganesh as all data flows through the proxy. As our focus was on the WAN link between the application server and the database, there were no constraints on the link between the simulated test clients and the web server.

5. THROUGHPUT AND RESPONSE TIME

In this section, we address the first question raised in Section 4: Can performance can be improved significantly by exploiting similarity across database results? To answer this question, we use results from the BBOARD and AUCTION benchmarks. We use two metrics to quantify the performance improvement obtainable through the use of Ganesh: throughput, from the perspective of the web server, and average response time, from the perspective of the client. Throughput is measured in terms of the number of client requests that can be serviced per second.

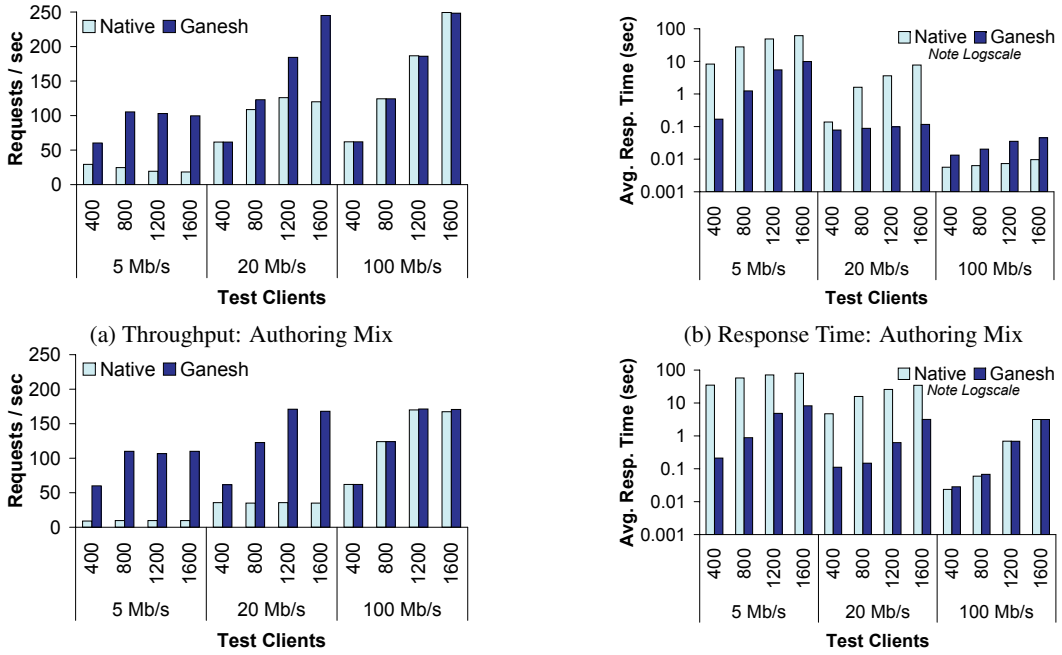
5.1 BBOARD Results and Analysis

5.1.1 Authoring Mix

Figures 6 (a) and (b) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients for BBOARD's Authoring Mix.

As Figure 6 (a) shows, Native easily saturates the 5 Mb/s link. At 400 clients, the Native solution delivers 29 requests/sec with an average response time of 8.3 seconds. Native's throughput drops with an increase in test clients as clients timeout due to congestion at the application server. Usability studies have shown that response times above 10 seconds cause the user to move on to

[1]As Java lacks a `sizeof()` operator, Java caches therefore limit their size based on the number of objects. The size of cache dumps taken at the end of the experiments never exceeded 212 MB.



(a) Throughput: Authoring Mix

(b) Response Time: Authoring Mix

(c) Throughput: Browsing Mix

(d) Response Time: Browsing Mix

Mean of three trials. The maximum standard deviation for throughput and response time was 9.8% and 11.9% of the corresponding mean.

Figure 6: BBOARD Benchmark - Throughput and Average Response Time

other tasks [24]. Based on these numbers, increasing the number of test clients makes the Native system unusable. Ganesh at 5 Mb/s, however, delivers a twofold improvement with 400 test clients and a fivefold improvement at 1200 clients. Ganesh's performance drops slightly at 1200 and 1600 clients as the network is saturated. Compared to Native, Figure 6 (b) shows that Ganesh's response times are substantially lower with sub-second response times at 400 clients.

Figure 6 (a) also shows that for 400 and 800 test clients Ganesh at 5 Mb/s has the same throughput and average response time as Native at 20 Mb/s. Only at 1200 and 1600 clients does Native at 20 Mb/s deliver higher throughput than Ganesh at 5 Mb/s.

Comparing both Ganesh and Native at 20 Mb/s, we see that Ganesh is no longer bandwidth constrained and delivers up to a twofold improvement over Native at 1600 test clients. As Ganesh does not saturate the network with higher test client configurations, at 1600 test clients, its average response time is 0.1 seconds rather than Native's 7.7 seconds.

As expected, there are no visible gains from Ganesh at the higher bandwidth of 100 Mb/s where the network is no longer the bottleneck. Ganesh, however, still tracks Native in terms of throughput.

5.1.2 *Browsing Mix*

Figures 6 (c) and (d) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients for BBOARD's Browsing Mix.

Regardless of the test client configuration, Figure 6 (c) shows that Native's throughput at 5 Mb/s is limited to 10 reqs/sec. Ganesh at 5 Mb/s with 400 test clients, delivers more than a sixfold increase in throughput. The improvement increases to over a elevenfold increase at 800 test clients before Ganesh saturates the network. Further, Figure 6 (d) shows that Native's average response time of 35 seconds at 400 test clients make the system unusable. These high response times further increase with the addition of test clients. Even with the 1600 test client configuration Ganesh delivers an acceptable average response time of 8.2 seconds.

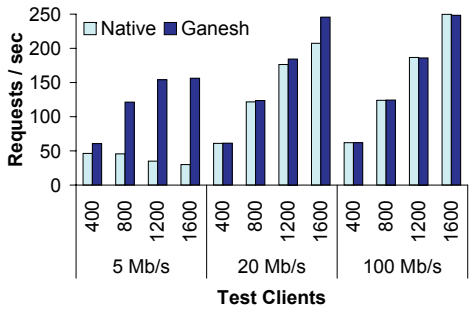
Due to the data-intensive nature of the Browsing mix, Ganesh at 5 Mb/s surprisingly performs much better than Native at 20 Mb/s. Further, as shown in Figure 6 (d), while the average response time for Native at 20 Mb/s is acceptable at 400 test clients, it is unusable with 800 test clients with an average response time of 15.8 seconds. Like the 5 Mb/s case, this response time increases with the addition of extra test clients.

Ganesh at 20 Mb/s and both Native and Ganesh at 100 Mb/s are not bandwidth limited. However, performance plateaus out after 1200 test clients due to the database CPU being saturated.

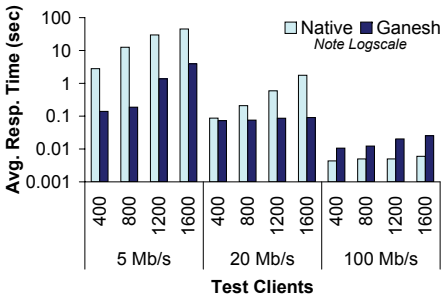
5.1.3 *Filter Variant*

We were surprised by the Native performance from the BBOARD benchmark. At the bandwidth of 5 Mb/s, Native performance was lower than what we had expected. It turned out the benchmark code that displays stories read all the comments associated with the particular story from the database and only then did some post-processing to select the comments to be displayed. While this is exactly the behavior of SlashCode, the code base behind the Slashdot web site, we decided to modify the benchmark to perform some pre-filtering at the database. This modified benchmark, named the *Filter Variant*, models a developer who applies optimizations at the SQL level to transfer less data. In the interests of brevity, we only briefly summarize the results from the Authoring mix.

For the Authoring mix, at 800 test clients at 5 Mb/s, Figure 7 (a) shows that Native's throughput increase by 85% when compared to the original benchmark while Ganesh's improvement is smaller at 15%. Native's performance drops above 800 clients as the test clients time out due to high response times. The most significant gain for Native is seen at 20 Mb/s. At 1600 test clients, when compared to the original benchmark, Native sees a 73% improvement in throughput and a 77% reduction in average response time. While



(a) Throughput: Authoring Mix



(b) Response Time: Authoring Mix

Mean of three trials. The maximum standard deviation for throughput and response time was 7.2% and 11.5% of the corresponding mean.

Figure 7: BBOARD Benchmark - Filter Variant - Throughput and Average Response Time

Ganesh sees no improvement when compared to the original, it still processes 19% more requests/sec than Native. Thus, while the optimizations were more helpful to Native, Ganesh still delivers an improvement in performance.

5.2 AUCTION Results and Analysis

5.2.1 Bidding Mix

Figures 8 (a) and (b) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients for AUCTION's Bidding Mix. As mentioned earlier, the Bidding mix consists of a mixture of read and write operations.

The AUCTION benchmark is not as data intensive as BBOARD. Therefore, most of the gains are observed at the lower bandwidth of 5 Mb/s. Figure 8 (a) shows that the increase in throughput due to Ganesh ranges from 8% at 400 test clients to 18% with 1600 test clients. As seen in Figure 8 (b), the average response times for Ganesh are significantly lower than Native ranging from a decrease of 84% at 800 test clients to 88% at 1600 test clients.

Figure 8 (a) also shows that with a fourfold increase of bandwidth from 5 Mb/s to 20 Mb/s, Native is no longer bandwidth constrained and there is no performance difference between Ganesh and Native. With the higher test client configurations, we did observe that the bandwidth used by Ganesh was lower than Native. Ganesh might still be useful in these non-constrained scenarios if bandwidth is purchased on a metered basis. Similar results are seen for the 100 Mb/s scenario.

5.2.2 Browsing Mix

For AUCTION's Browsing Mix, Figures 8 (c) and (d) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients.

Again, most of the gains are observed at lower bandwidths. At 5 Mb/s, Native and Ganesh deliver similar throughput and response times with 400 test clients. While the throughput for both remains the same at 800 test clients, Figure 8 (d) shows that Ganesh's average response time is 62% lower than Native. Native saturates the link at 800 clients and adding extra test clients only increases the average response time. Ganesh, regardless of the test client configuration, is not bandwidth constrained and maintains the same response time. At 1600 test clients, Figure 8 (c) shows that Ganesh's throughput is almost twice that of Native.

At the higher bandwidths of 20 and 100 Mb/s, neither Ganesh nor Native is bandwidth limited and deliver equivalent throughput and response times.

Benchmark	Orig. Size	Ganesh Size	Rabin Size
SelectSort$_1$	223.6 MB	5.4 MB	219.3 MB
SelectSort$_2$	223.6 MB	5.4 MB	223.6 MB

Table 2: Similarity Microbenchmarks

6. STRUCTURAL VS. RABIN SIMILARITY

In this section, we address the second question raised in Section 4: How important is Ganesh's structural similarity detection relative to Rabin fingerprinting-based similarity detecting? To answer this question, we used microbenchmarks and the BBOARD and AUCTION benchmarks. As Ganesh always performed better than Rabin fingerprinting, we only present a subset of the results here in the interests of brevity.

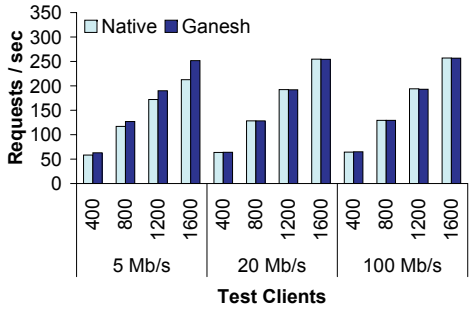
6.1 Microbenchmarks

Two microbenchmarks show an example of the effects of data reordering on Rabin fingerprinting algorithm. In the first microbenchmark, *SelectSort*$_1$, a query with a specified sort order selects 223.6 MB of data spread over approximately 280 K rows. The query is then repeated with a different sort attribute. While the same number of rows and the same data is returned, the order of rows is different. In such a scenario, one would expect a large amount of similarity to be detected between both results. As Table 2 shows, Ganesh's row-based algorithm achieves a 97.6% reduction while the Rabin fingerprinting algorithm, with the average chunk size parameter set to 4 KB, only achieves a 1% reduction. The reason, as shown earlier in Figure 2, is that with Rabin fingerprinting, the spans of data between two consecutive boundaries usually cross row boundaries. With the order of the rows changing in the second result and the Rabin fingerprints now spanning different rows, the algorithm is unable to detect significant similarity. The small gain seen is mostly for those single rows that are large enough to be broken into multiple chunks.

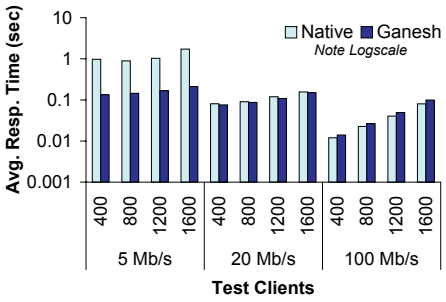
SelectSort$_2$, another micro-benchmark executed the same queries but increased the minimum chunk size of the Rabin fingerprinting algorithm. As can be seen in Table 2, even the small gain from the previous microbenchmark disappears as the minimum chunk size was greater than the average row size. While one can partially address these problems by dynamically varying the parameters of the Rabin fingerprinting algorithm, this can be computationally expensive, especially in the presence of changing workloads.

6.2 Application Benchmarks

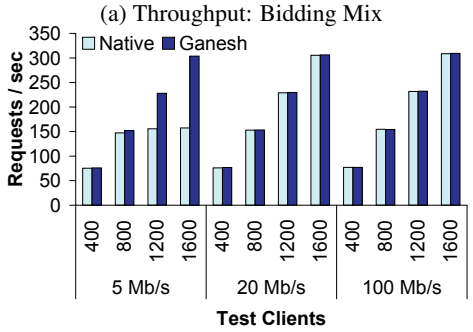
We ran the BBOARD benchmark described in Section 4.1.1 on two versions of Ganesh: the first with Rabin fingerprinting used as



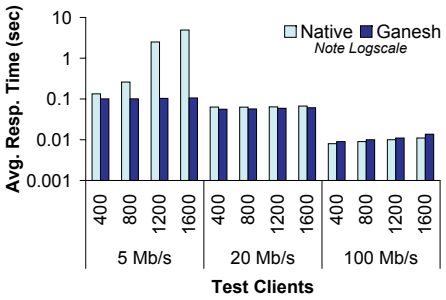
(a) Throughput: Bidding Mix



(b) Response Time: Bidding Mix



(c) Throughput: Browsing Mix



(d) Response Time: Browsing Mix

Mean of three trials. The maximum standard deviation for throughput and response time was 2.2% and 11.8% of the corresponding mean.

Figure 8: `AUCTION` Benchmark - Throughput and Average Response Time

the chunking algorithm and the second with Ganesh's row-based algorithm. Rabin's results for the Browsing Mix are normalized to Ganesh's results and presented in Figure 9.

As Figure 9 (a) shows, at 5 Mb/s, independent of the test client configuration, Rabin significantly underperforms Ganesh. This happens because of a combination of two reasons. First, as outlined in Section 3.1, Rabin finds less similarity as it does not exploit the result's structural information. Second, this benchmark contained some queries that generated large results. In this case, Rabin, with a small average chunk size, generated a large number of objects that evicted other useful data from the cache. In contrast, Ganesh was able to detect these large rows and correspondingly increase the size of the chunks. This was confirmed as cache statistics showed that Ganesh's hit ratio was roughly three time that of Rabin. Throughput measurements at 20 Mb/s were similar with the exception of Rabin's performance with 400 test clients. In this case, Ganesh was not network limited and, in fact, the throughput was the same as 400 clients at 5 Mb/s. Rabin, however, took advantage of the bandwidth increase from 5 to 20 Mb/s to deliver a slightly better performance. At 100 Mb/s, Rabin's throughput was almost similar to Ganesh as bandwidth was no longer a bottleneck.

The normalized response time, presented in Figure 9 (b), shows similar trends. At 5 and 20 Mb/s, the addition of test clients decreases the normalized response time as Ganesh's average response time increases faster than Rabin's. However, at no point does Rabin outperform Ganesh. Note that at 400 and 800 clients at 100 Mb/s, Rabin does have a higher overhead even when it is not bandwidth constrained. As mentioned in Section 3.1, this is due to the fact that Rabin has to hash each `ResultSet` twice. The overhead disappears with 1200 and 1600 clients as the database CPU is saturated and limits the performance of both Ganesh and Rabin.

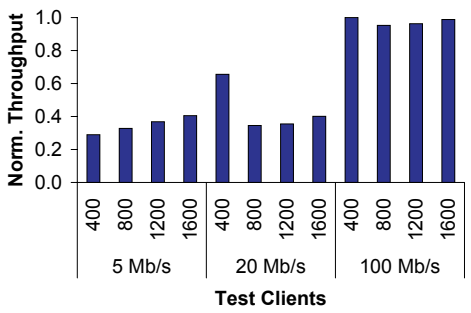
7. PROXY OVERHEAD

In this section, we address the third question raised in Section 4: Is the overhead of Ganesh's proxy-based design acceptable? To answer this question, we concentrate on its performance at the higher bandwidths. Our evaluation in Section 5 showed that Ganesh, when compared to Native, can deliver a substantial throughput improvement at lower bandwidths. It is only at higher bandwidths that latency, measured by the average response time for a client request, and throughput, measured by the number of client requests that can be serviced per second, overheads would be visible.

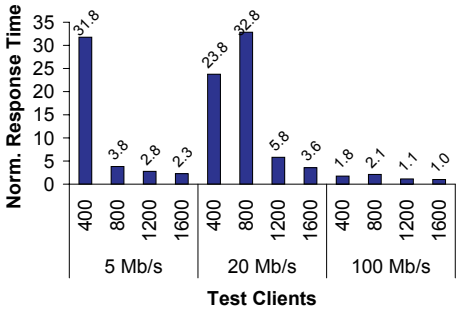
Looking at the Authoring mix of the original `BBOARD` benchmark, there are no visible gains from Ganesh at 100 Mb/s. Ganesh, however, still tracks Native in terms of throughput. While the average response time is higher for Ganesh, the absolute difference is in between 0.01 and 0.04 seconds and would be imperceptible to the end-user. The Browsing mix shows an even smaller difference in average response times. The results from the filter variant of the `BBOARD` benchmarks are similar. Even for the `AUCTION` benchmark, the difference between Native and Ganesh's response time at 100 Mb/s was never greater than 0.02 seconds. The only exception to the above results was seen in the filter variant of the `BBOARD` benchmark where Ganesh at 1600 test clients added 0.85 seconds to the average response time. Thus, even for much faster networks where the WAN link is not the bottleneck, Ganesh always delivers throughput equivalent to Native. While some extra latency is added by the proxy-based design, it is usually imperceptible.

8. RELATED WORK

To the best of our knowledge, Ganesh is the first system that combines the use of hash-based techniques with caching of database results to improve throughput and response times for applications with dynamic content. We also believe that it is also the first system to demonstrate the benefits of using structural information for



(a) Normalized Throughput: Higher is better



(b) Normalized Response Time: Higher is worse

For throughput, a normalized result greater than 1 implies that Rabin is better, For response time, a normalized result greater than 1 implies that Ganesh is better. Mean of three trials. The maximum standard deviation for throughput and response time was 9.1% and 13.9% of the corresponding mean.

Figure 9: Normalized Comparison of Ganesh vs. Rabin - BBOARD Browsing Mix

detecting similarity. In this section, we first discuss alternative approaches to caching dynamic content and then examine other uses of hash-based primitives in distributed systems.

8.1 Caching Dynamic Content

At the database layer, a number of systems have advocated middle-tier caching where parts of the database are replicated at the edge or server [3, 4, 20]. These systems either cache entire tables in what is essentially a replicated database or use materialized views from previous query replies [19]. They require tight integration with the back-end database to ensure a time bound on the propagation of updates. These systems are also usually targeted towards workloads that do not require strict consistency and can tolerate stale data. Further, unlike Ganesh, some of these mid-tier caching solutions [2, 3], suffer from the complexity of having to participate in query planing and distributed query processing.

Gao et al. [15] propose using a distributed object replication architecture where the data store's consistency requirements are adapted on a per-application basis. These solutions require substantial developer resources and detailed understanding of the application being modified. While systems that attempt to automate the partitioning and replication of an application's database exist [34], they do not provide full transaction semantics. In comparison, Ganesh does not weaken any of the semantics provided by the underlying database.

Recent work in the evaluation of edge caching options for dynamic web sites [38] has suggested that, without careful planning, employing complex offloading strategies can hurt performance. Instead, the work advocates for an architecture in which all tiers except the database should be offloaded to the edge. Our evaluation of Ganesh has shown that it would benefit these scenarios. To improve database scalability, C-JDBC [10], SSS [22], and Ganymed [28] also advocate the use of an interposition-based architecture to transparently cluster and replicate databases at the middleware level. The approaches of these architectures and Ganesh are complementary and they would benefit each other.

Moving up to the presentation layer, there has been widespread adoption of fragment-based caching [14], which improves cache utilization by separately caching different parts of generated web pages. While fragment-based caching works at the edge, a recent proposal has proposed moving web page assembly to the clients to optimize content delivery [31]. While Ganesh is not used at the presentation layer, the same principles have been applied in Duplicate Transfer Detection [25] to increase web cache efficiency as well as for web access across bandwidth limited links [33].

8.2 Hash-based Systems

The past few years have seen the emergence of many systems that exploit hash-based techniques. At the heart of all these systems is the idea of detecting similarity in data without requiring interpretation of that data. This simple yet elegant idea relies on cryptographic hashing, as discussed earlier in Section 2. Successful applications of this idea span a wide range of storage systems. Examples include peer-to-peer backup of personal computing files [11], storage-efficient archiving of data [29], and finding similar files [21].

Spring and Wetherall [35] apply similar principles at the network level. Using synchronized caches at both ends of a network link, duplicated data is replaced by smaller tokens for transmission and then restored at the remote end. This and other hash-based systems such as the CASPER [37] and LBFS [26] filesystems, and Layer-2 bandwidth optimizers such as Riverbed and Peribit use Rabin fingerprinting [30] to discover spans of commonality in data. This approach is especially useful when data items are modified *in-place* through insertions, deletions, and updates. However, as Section 6 shows, the performance of this technique can show a dramatic drop in the presence of data reordering. Ganesh instead uses row boundaries as dividers for detecting similarity.

The most aggressive use of hash-based techniques is by systems that use hashes as the primary identifiers for objects in persistent storage. Storage systems such as CFS [12] and PAST [13] that have been built using distributed hash tables fall into this category. Single Instance Storage [6] and Venti [29] are other examples of such systems. As discussed in Section 2.2, the use of cryptographic hashes for addressing persistent data represents a deeper level of faith in their collision-resistance than that assumed by Ganesh. If time reveals shortcomings in the hash algorithm, the effort involved in correcting the flaw is much greater. In Ganesh, it is merely a matter of replacing the hash algorithm.

9. CONCLUSION

The growing use of dynamic web content generated from relational databases places increased demands on WAN bandwidth. Traditional caching solutions for bandwidth and latency reduction are often ineffective for such content. This paper shows that the impact of WAN accesses to databases can be substantially reduced through the Ganesh architecture without any compromise of the database's strict consistency semantics. The essence of the Ganesh architecture is the use of computation at the edges to reduce communication through the Internet. Ganesh is able to use cryptographic hashes to detect similarity with previous results and send

compact recipes of results rather than full results. Our design uses interposition to achieve complete transparency: clients, application servers, and database servers are all unaware of Ganesh's presence and require no modification.

Our experimental evaluation confirms that Ganesh, while conceptually simple, can be highly effective in improving throughput and response time. Our results also confirm that exploiting the structure present in database results to detect similarity is crucial to this performance improvement.

10. REFERENCES

[1] AKELLA, A., SESHAN, S., AND SHAIKH, A. An empirical evaluation of wide-area internet bottlenecks. In *Proc. 3rd ACM SIGCOMM Conference on Internet Measurement* (Miami Beach, FL, USA, Oct. 2003), pp. 101–114.

[2] ALTINEL, M., BORNHÖVD, C., KRISHNAMURTHY, S., MOHAN, C., PIRAHESH, H., AND REINWALD, B. Cache tables: Paving the way for an adaptive database cache. In *Proc. of 29th VLDB* (Berlin, Germany, 2003), pp. 718–729.

[3] ALTINEL, M., LUO, Q., KRISHNAMURTHY, S., MOHAN, C., PIRAHESH, H., LINDSAY, B. G., WOO, H., AND BROWN, L. Dbcache: Database caching for web application servers. In *Proc. 2002 ACM SIGMOD* (2002), pp. 612–612.

[4] AMIRI, K., PARK, S., TEWARI, R., AND PADMANABHAN, S. Dbproxy: A dynamic data cache for web applications. In *Proc. IEEE International Conference on Data Engineering (ICDE)* (Mar. 2003).

[5] BLACK, J. Compare-by-hash: A reasoned analysis. In *Proc. 2006 USENIX Annual Technical Conference* (Boston, MA, May 2006), pp. 85–90.

[6] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., , AND DOUCEUR, J. R. Single instance storage in windows 2000. In *Proc. 4th USENIX Windows Systems Symposium* (Seattle, WA, Aug. 2000), pp. 13–24.

[7] BREWER, E. A. Lessons from giant-scale services. *IEEE Internet Computing 5*, 4 (2001), 46–55.

[8] BRODER, A., GLASSMAN, S., MANASSE, M., AND ZWEIG, G. Syntactic clustering of the web. In *Proc. 6th International WWW Conference* (1997).

[9] CECCHET, E., CHANDA, A., ELNIKETY, S., MARGUERITE, J., AND ZWAENEPOEL, W. Performance comparison of middleware architectures for generating dynamic web content. In *Proc. Fourth ACM/IFIP/USENIX International Middleware Conference* (Rio de Janeiro, Brazil, June 2003).

[10] CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. C-JDBC: Flexible database clustering middleware. In *Proc. 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004).

[11] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making backup cheap and easy. In *OSDI: Symposium on Operating Systems Design and Implementation* (2002).

[12] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles* (Banff, Canada, Oct. 2001).

[13] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII* (Schloss Elmau, Germany, May 2001), pp. 75–80.

[14] Edge side includes. http://www.esi.org.

[15] GAO, L., DAHLIN, M., NAYATE, A., ZHENG, J., AND IYENGAR, A. Application specific data replication for edge services. In *WWW '03: Proc. Twelfth International Conference on World Wide Web* (2003), pp. 449–460.

[16] HEMMINGER, S. Netem - emulating real networks in the lab. In *Proc. 2005 Linux Conference Australia* (Canberra, Australia, Apr. 2005).

[17] HENSON, V. An analysis of compare-by-hash. In *Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (May 2003), pp. 13–18.

[18] Jmob benchmarks. `http://jmob.objectweb.org/`.

[19] LABRINIDIS, A., AND ROUSSOPOULOS, N. Balancing performance and data freshness in web database servers. In *Proc. 29th VLDB Conference* (Sept. 2003).

[20] LARSON, P.-A., GOLDSTEIN, J., AND ZHOU, J. Transparent mid-tier database caching in sql server. In *Proc. 2003 ACM SIGMOD* (2003), pp. 661–661.

[21] MANBER, U. Finding similar files in a large file system. In *Proc. USENIX Winter 1994 Technical Conference* (San Fransisco, CA, 17–21 1994), pp. 1–10.

[22] MANJHI, A., AILAMAKI, A., MAGGS, B. M., MOWRY, T. C., OLSTON, C., AND TOMASIC, A. Simultaneous scalability and security for data-intensive web applications. In *Proc. 2006 ACM SIGMOD* (June 2006), pp. 241–252.

[23] MENEZES, A. J., VANSTONE, S. A., AND OORSCHOT, P. C. V. *Handbook of Applied Cryptography*. CRC Press, 1996.

[24] MILLER, R. B. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference* (1968), pp. 267–277.

[25] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in http. In *Proc. First Symposium on Networked Systems Design and Implementation* (San Francisco, CA, Mar. 2004).

[26] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, Oct. 2001).

[27] PFEIFER, D., AND JAKSCHITSCH, H. Method-based caching in multi-tiered server applications. In *Proc. Fifth International Symposium on Distributed Objects and Applications* (Catania, Sicily, Italy, Nov. 2003).

[28] PLATTNER, C., AND ALONSO, G. Ganymed: Scalable replication for transactional web applications. In *Proc. 5th ACM/IFIP/USENIX International Conference on Middleware* (2004), pp. 155–174.

[29] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proc. FAST 2002 Conference on File and Storage Technologies* (2002).

[30] RABIN, M. Fingerprinting by random polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81* (1981).

[31] RABINOVICH, M., XIAO, Z., DOUGLIS, F., AND KALMANEK, C. Moving edge side includes to the real edge – the clients. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, Mar. 2003).

[32] REESE, G. *Database Programming with JDBC and Java*, 1st ed. O'Reilly, June 1997.

[33] RHEA, S., LIANG, K., AND BREWER, E. Value-based web caching. In *Proc. Twelfth International World Wide Web Conference* (May 2003).

[34] SIVASUBRAMANIAN, S., ALONSO, G., PIERRE, G., AND VAN STEEN, M. Globedb: Autonomic data replication for web applications. In *WWW '05: Proc. 14th International World-Wide Web conference* (May 2005).

[35] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *Proc. of ACM SIGCOMM* (Aug. 2000).

[36] TOLIA, N., HARKES, J., KOZUCH, M., AND SATYANARAYANAN, M. Integrating portable and distributed storage. In *Proc. 3rd USENIX Conference on File and Storage Technologies* (San Francisco, CA, Mar. 2004).

[37] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., PERRIG, A., AND BRESSOUD, T. Opportunistic use of content addressable storage for distributed file systems. In *Proc. 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003), pp. 127–140.

[38] YUAN, C., CHEN, Y., AND ZHANG, Z. Evaluation of edge caching/offloading for dynamic content delivery. In *WWW '03: Proc. Twelfth International Conference on World Wide Web* (2003), pp. 461–471.