Vrije Universiteit Amsterdam
Faculty of sciences
Department of computer science

vrije Universiteit amsterdam

Scalability of Web Applications in CDN Environments

Thesis: **Master of Science**

Specialization: **Parallel and Distributed Computer Science**

Tobias Groothuyse

Student number: 1278983

August 2006

Supervisors:

Dr. Guillaume Pierre

Swaminathan Sivasubramanian

Contents

Chapter 1 Introduction **1**

Chapter 2 Related work **4**

2.1 Fragment caching 4

2.2 Edge server computing 4

2.3 Query caching 4

2.3.1 Content-aware query caching 5

2.3.2 Content-blind query caching 5

2.4 Database replication 6

2.4.1 Two phase locking 6

2.4.2 Snapshot isolation 6

2.5 Adaptive replication 7

Chapter 3 Query caching **8**

3.1 Cache replacement policy 8

3.2 Cache size 8

3.2.1 Distributed caching 9

3.2.2 Hierarchical caching 9

3.3 A combined approach 9

Chapter 4 Database replication **11**

4.1 Proposed algorithm 12

4.1.1 Table cluster construction 12

4.1.2 Table cluster cost estimation 12

4.1.3 Database node load estimation 13

4.1.4 Placement algorithm 13

4.2 Query routing 16

4.2.1 Round robin per QID 16

4.2.2 Cost-based 16

Chapter 5 Evaluation **17**

5.1 Experimental setup 17

5.2 Query caching 18

5.3 Query cost estimation 20

5.4 Database replication 21

5.5 Combined approach 23

Chapter 6 Conclusion.......... 25

Abstract

The scalability bottleneck of data-intensive Web applications often turns out to be the database. This bottleneck can be alleviated by decreasing the load posed on the database or increasing the database throughput. Existing solutions are either based on query caching that intend to decrease the database load or database replication that intend to increase the throughput of the database. However, as we show in this thesis, none of these solutions provide sufficient scalability for demanding applications.

This thesis combines three approaches in order to alleviate the database bottleneck. We first study the impact of distributed and hierarchical organizations for database query caches. Second, we propose a novel data replication technique which improves the throughput compared to a fully replicated database. Third, we introduce a cost-based query routing policy to improve the load balance of the database nodes at run time. All approaches exploit the fact that an application's query workload is based on a small set of read and write templates. Our evaluations show that a combination of distributed and hierarchical caching can result in hit ratios up to 60%. Furthermore, we show that both the novel replication technique as well as the cost-based routing can significantly increase the scalability. Finally, our evaluations show that for a fixed workload a combined approach using distributed and hierarchical caching as well as the novel data placement and cost-based routing can process 95% of all queries within 100ms compared to only 1% for a fully replicated database.

Chapter 1

Introduction

Since the breakthrough of the World Wide Web as one the most important sources of information, numerous companies have seen a significant increase in their website traffic. Most of these companies have a commercial interest to make sure that all their visitors get the best possible service. However, for websites that receive a lot of traffic, single-server hosting solutions can no longer provide the needed capacity.

Content Delivery Networks (CDNs) such as Akamai specialize in serving Web pages for companies that attract a lot of visitors [22]. A Content Delivery Network consists of multiple clusters of edge servers (see Figure 1). Clusters are located at different geographical locations across the globe. Visitors of a website hosted by a CDN are redirected to the closest edge server. An edge server receives pages from the customer's origin Web server and in turn serves them to clients.

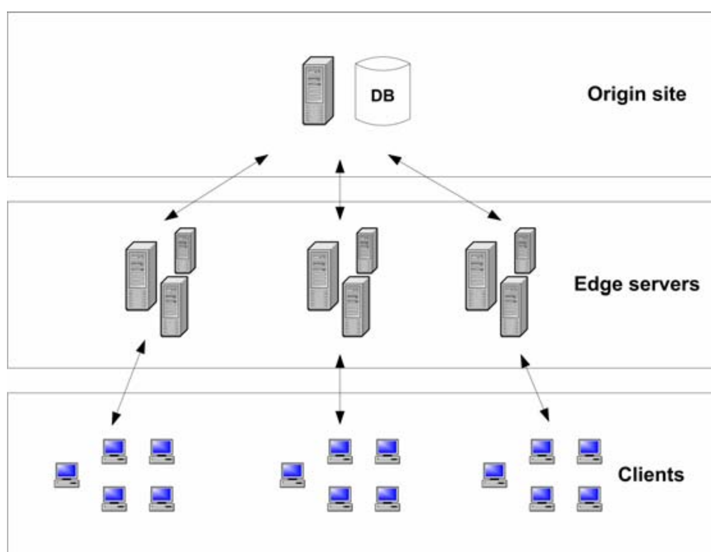

Figure 1. Edge server computing infrastructure

CDNs perform well when the content they are hosting consists of static pages that are infrequently updated. However, current day Web sites almost always generate pages dynamically using Web applications according to user preferences and request parameters which makes traditional CDN techniques insufficient. One solution is to divide the generated pages into fragments. The fragments can then be cached at edge servers. Unfortunately, fragment caching works well only if the underlying data that is used to generate the fragments is not frequently updated.

Edge server computing is a different approach that allows the data to be updated more frequently. In edge server computing the application code is replicated at all edge servers. Each edge server thus becomes capable of generating any page. While this approach works well for applications that are not data-intensive, in other cases the database quickly becomes the performance bottleneck. As all edge servers issue their database requests to the same database server, this database server will at a certain point become

overloaded. Besides the performance bottleneck, edge server computing also increases the client perceived latency because each request for a page will most likely involve one or more database queries which all incur wide-area latency.

To allow data-intensive Web applications to scale, either the throughput of the database will have to increase or the incoming workload of the database will have to decrease. To decrease the incoming workload of the database a query cache can be placed at each edge server to handle a portion of the queries issued from the application, thereby offloading the database server. To increase the capacity of the database, one can replicate the database. In a replicated database, all queries are distributed between the database replicas, thereby increasing the maximum throughput.

All current approaches use only a single technique to improve the scalability. However, a single technique only addresses a single aspect of the performance bottleneck that is currently limiting the scalability of Web applications. Consequently, a single technique will at a certain point no longer increase the scalability. In contrast, this thesis investigates the simultaneous use of several techniques to improve the scalability of Web applications in CDN environments.

The basic model adopted in this thesis is outlined in figure 2. The contributions proposed in this thesis are threefold. The first contribution analyzes the impact of different cache organization strategies on the database workload. This is done by adding two levels of query caches between the edge servers and the database server. At each cluster of edge servers a level one (L1) query cache is placed. Furthermore a second level (L2) cache is placed at the origin site such that database queries that cannot be answered by an L1 cache are forwarded to the L2 cache. The L1 caches will exploit the temporal locality present in the query workload. However, because the content of each L1 cache will have considerable overlap, invalidations on the L1 caches will subsequently cause them to send identical requests to the L2 cache. This remaining locality can then be exploited by the L2 cache. Our evaluations show that for the studied application the L1 caches can get hit ratios of up to 44%. Furthermore, the L2 cache can get an additional hit ratio of up to 15%.

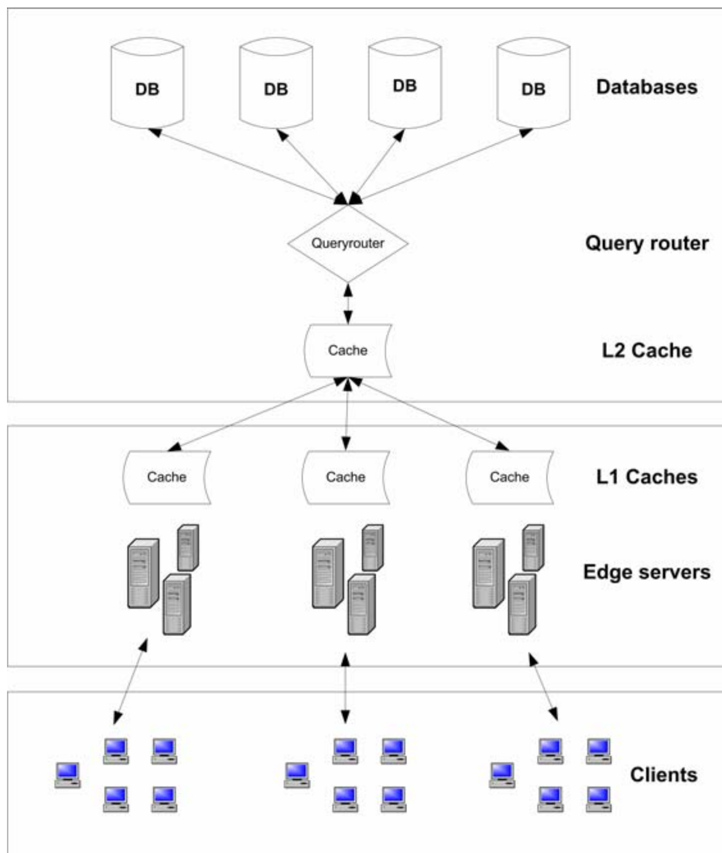

Figure 2. System model

The second contribution of this thesis consists of a novel database replication algorithm that uses knowledge about the application to compute data placements likely to provide excellent throughput. Replicating data on multiple database nodes has a positive and a negative effect. The positive effect is that read queries can be processed by more than one node, increasing the throughput. The negative effect is that the replicated data needs to be maintained consistent by applying the write queries that modify the data at all nodes at which the data is replicated. Our algorithm computes an efficient data placement by balancing these two effects in such a way that the throughput is maximized. Evaluations show that the data placement computed by this algorithm can increase the throughput from 200% up to 700% compared to a fully replicated database that uses the same number of storage nodes.

A crucial aspect of the performance of a replicated database is the routing policy adopted by the query router. In a partially replicated database, queries can no longer be sent to arbitrary database nodes. Read queries can only be executed on nodes which store the needed data and write queries must be executed on all nodes which store the data that they modify. Therefore the third contribution of this thesis consists of adding a more sophisticated routing algorithm to the query router which improves the load balancing of the database servers by taking the execution cost of individual queries into account. Performance evaluations show that cost-based routing can improve the throughput up to 200% in cases where the execution costs of individual queries are very different.

The rest of this thesis is organized as follows. Chapter two describes the related work on CDNs, query caching and database replication. Chapter three describes the details of the query caches. Chapter four describes the database replication method and the algorithms used in the query router. Chapter five discusses the performance of the different parts of the system. Chapter six concludes this thesis.

Chapter 2

Related work

A number of different approaches exist which can be used to increase the scalability of Web applications within the context of content delivery networks (CDNs).

2.1 Fragment caching

In fragment caching, each Web page is broken down into fragments [20, 21]. Fragments independently consist of static data but sometimes they can also consist of dynamic data. The fragments are cached at edge servers and are used by the edge servers to generate pages upon client requests. When the underlying data of a fragment is modified it has to be removed from all caches in order to maintain consistency. Fragment caching performs well in situations where the temporal locality of the requests is high and the update rate of the database is low. For applications that do not meet these conditions, fragment caching does not perform well.

2.2 Edge server computing

Instead of caching fragments at the edge servers, edge server computing replicates the application code at the edge servers (see figure 1). This allows pages to be generated close to the client. Replicating application code is relatively easy and scales up the application tier very well. However simple code replication still requires all database queries to be forwarded to the centralized database server. The number of edge servers can be increased to cope with an increasing Web page request rate. However, this leads to an increased query request rate, which will at a certain point overload the database. Thus, the database becomes the bottleneck.

2.3 Query caching

Query caching is a technique that aims to alleviate the database bottleneck in edge server computing. Query caching allows data stored in the database, to be cached at locations closer to the client. This allows edge servers to answer queries locally when all required data is present. If the data is not present, the query is forwarded to the database. The effectiveness of database offloading depends on the hit ratio of the cache.

Caching can be done at various levels to achieve different performance optimizations. The caches can be placed close to the edge servers, close to the database or somewhere in between. Caches that are placed closer to the edge servers will result in lower client perceived latency on a cache hit. However keeping the cache consistent will result in increased WAN traffic. Caches that are placed close to the database do not incur any WAN traffic to maintain consistency but will result in a higher client perceived latency.

Query caching solutions can be classified into to approaches. The first one is content-aware query caching. Content-aware query caching merges and stores query results together and analyzes the information contained in the query to determine whether it can be answered from the cache. The second one is content-blind query caching. Content-blind query caching stores query results independently and does not use any information contained in the query to determine whether it can be answered from the cache.

2.3.1 Content-aware query caching

Content-aware query caching systems run a complete DBMS at each cache. The DBMS is used to store the data that is being cached. Content-aware query caching systems can be divided into two categories. The first category contains systems that cache complete or partial tables from the database server, whereas systems in the second category cache the result of executed queries.

The first category of systems that cache complete or partial tables implement this functionality by using materialized views. A materialized view consists of all the tuples that are represented by the view. To maintain consistency, caches register to an update stream from the database server for the cached materialized views. Updates to tuples that are present in the cached views will then be forwarded from the database server to the caches. Queries that only use data which is available in the local materialized views can be answered by the cache. If data is used that is not present in any locally cached view, the query is forwarded to the database server. Update, Delete and Insert (UDI) queries are always forwarded to the database server, where they will execute and possibly trigger an update to the caches. Two systems that follow this approach are DBCache [6] and MTCache [7]. Caching complete or partial tables offers a simple, yet powerful caching mechanism. However, there are two scalability issues. The first one is that the materialized views are defined by the system administrator at system initialization time. This prohibits the cache to dynamically adapt to changing workloads, which can result in poor hit ratios over time. The second issue is that the granularity of the cached data is very coarse-grained. The amount of update traffic will therefore be considerable and a lot of data will be cached that is never actually used to answer queries.

The second category of systems merge and store results from different queries together in the DBMS. To determine whether an incoming query can be answered from the cache a query containment check has to be performed. Query containment is necessary because the needed data is not necessarily available in the cache. If a query cannot be answered from the cache it is forwarded to the origin database. Examples of systems that fall into this category are DBProxy [4] and a system developed by Amza et. al. [5] (called Amza from this point on).

UDI queries modify the database and should therefore invalidate some of the query results maintained in the caches. In Amza, the cache is maintained consistent by removing all tuples from the cache which have become obsolete due to a UDI query. Unlike the method used in Amza, DBProxy maintains consistency by using continuous updates. In this case, each cache subscribes to updates from the database server for those tuples which it has stored in the cache.

Removing tuples from the cache, as done in Amza, reduces the hit ratio until the moment the cache is full and warmed up again. This has the negative effect that the request rate at the database server will increase, thereby limiting the scalability. The method applied by DBProxy does not suffer from a reduced cache hit ratio because entries are never invalidated. However, the downside of this approach is that WAN traffic will increase because some cached entries will be kept up to date which may never be used to answer future queries. Furthermore, the query containment check which is performed for each query can become a performance bottleneck under high load.

2.3.2 Content-blind query caching

Unlike the previous systems, GlobeCBC stores query results separately instead of merging them together[8]. In this case, a query can only be answered from the cache if the result of the exact same query is available in the cache. On the other hand, it does not need to perform any query analyzing or planning to determine whether a query can be answered from the cache.

GlobeCBC targets Web applications that do not require transactions, which is the case for the vast majority. This means that each request to the database consists of at most a single SQL query. An important observation about Web applications used in GlobeCBC is that in general the query workload is based on a set of query templates. A query template is a predefined SQL query in which a number of parameters can be differentiated, much like a prepared statement. For example: `SELECT * FROM t1 WHERE c1>@p1 AND c2=@p2`. In this example `@p1` and `@p2` are the formal parameters which are replaced by the actual parameters

at runtime. There are two types of templates: simple templates have a single parameter which has to be the primary key of the only table accessed in the query; complex templates are all templates which do not qualify as simple templates. Using this terminology, the query workload of a Web application can then be described as a set of N query templates. To allow GlobeCBC to refrain from query analyzing, the application developer will have to modify the application in such a way that each query issued to the database is accompanied with the associated Query template ID (QID) and the actual parameters. The QID and the actual parameters are used to index the cache for storage and invalidation purposes. Furthermore the application developer will have to supply a list of conflicts between the templates. If a query defined by a certain template consists of a UDI query then all other templates which should be invalidated (because they read from the data changed by this query) are listed as a conflicting templates.

Invalidation based on the conflicts between templates can be either coarse- or fine-grained. Coarse-grained invalidation causes all query results from conflicting templates to be invalidated upon arrival of the UDI query. This conservative approach invalidates a large number cached results. Fine-grained invalidation adds the refinement that query results of conflicting *simple* templates will only be invalidated in the case that the query parameter matches. The fine-grained invalidation method invalidates less cached results than the coarse-grained invalidation method and therefore results in a higher cache hit ratio and improved scalability.

2.4 Database replication

The most common approach to alleviate the database bottleneck is replication. Database replication however is complex due to consistency requirements. The level of consistency provided by the replication mechanisms and the method of maintaining that consistency largely determines the scalability of the system. This is due to the fact that the amount of effort that is spent on maintaining consistency directly relates to the level of consistency provided by the system. In the following, we classify replication systems based on the consistency mechanism they provide.

2.4.1 Two phase locking

The method that achieves the highest level of consistency is two phase locking (2PL). In 2PL the effect of a UDI query is synchronously applied at all replicas to ensure the state of all database nodes is always consistent. UDI queries can be submitted to any of the available nodes from which they are then synchronously applied using 2PL. Read queries are distributed between the available nodes. One of the systems that use 2PL to maintain consistency is MySQL Cluster [9].

For read dominant workloads this approach will be able to achieve considerable speedup. However for write dominant workloads, or workloads that have relatively a lot of write queries the situation is different. This is due to the fact that 2PL is a fairly expensive operation. Next to the fact that applying a UDI query costs much more then performing a read query, 2PL also has the negative property that UDI queries will block read queries that try to access the same data objects.

2.4.2 Snapshot isolation

Snapshot isolation (SI) is a multi-version concurrency control mechanism used in databases. Unlike 2PL it has the important property that read queries are never blocked by UDI queries. This mechanism is used in Ganymed [10] and [11]. The key idea in Ganymed is that all write queries are handled by a master node and that all read queries are handled by the other nodes, thereby offloading the master node. Read queries are applied under snapshot isolation to ensure they can not be blocked by ongoing write queries.

The major scalability issue of the Ganymed system is the master node. As soon as it cannot handle all the write transactions the system cannot scale any further. Other systems improve on Ganymed because instead of sending UDI queries to a single master node they can be distributed among all nodes [11, 12]. The middleware ensures that the state of all nodes is maintained consistent.

In traditional database systems both transaction ordering and durability are realized in a single action: the

writing of the commit record to disk. In middleware based database replication systems these tasks are naturally divided. The middleware determines the global ordering and the individual databases provide the durability by writing the commit record. Sameh et. al showed that this introduces a potentially very serious performance issue, since in worst case all writes have to be applied serially (instead of in bursts). They have developed the Tashkent system which provides three implementations of a replicated database [13]. The *Base* system is a standard replicated database using SI. This *Base* system is compared against *TashkentAPI*, a replicated database system implemented at the database level and *TashkentMW* implemented purely at the middleware level. Performance benchmarks including TPC-W and TPC-B show a significant throughput increase when comparing both *Tashkent* implementations against the *Base* implementation.

2.5 Adaptive replication

Adaptive replication systems either dynamically assign transactions or queries to a specific server, or dynamically vary the amount of database servers involved. This allows systems to cope with changes in the environment such as workload changes, changes in available resources and client distribution.

Systems that dynamically change the placement of data among the database nodes assign data-units to specific database nodes. A data-unit can be as fine-grained as a single table row but they can be clustered together to minimize the administrative overhead. A data-unit is assigned to one or more database nodes. One of the selected database nodes is assigned to be the master node. Master nodes are responsible for keeping the data-unit consistent at all replicas. The process of selecting nodes and masters is performed periodically in order to maintain an optimal configuration. The assignment of data-units to database nodes is done in such a way that a high percentage of database requests at each node can be answered from the locally available data-units. This adaptive replication technique is applied in GlobeDB [14] and Middle-R [15] and performs well for applications that have a high query locality and a relatively low update rate.

Besides optimizing the placement of data-units between the available database nodes another interesting approach is to look at the number of database servers. Ideally, we would like to minimize the amount of servers involved while still adhering to the different SLA's of the applications. Hosting multiple applications on a single database node will cause interference and competition for resources between the applications. However, it turns out that it is better to allow multiple applications to be hosted on the same set of database nodes in terms of the amount of SLA violations compared against a segregated approach. Two systems following this approach are presented in [16, 19].

Chapter 3

Query caching

This chapter introduces query caching techniques that can be used to alleviate the database bottleneck. These techniques will be evaluated in chapter five. The effectiveness of a query cache, which we define by its hit ratio, depends on two important factors: the cache replacement policy and the size of the cache. We also study the effect of distributed and hierarchical caching policies on the effectiveness of the system to offload the origin database.

3.1 Cache replacement policy

A cache replacement policy decides which items should be kept and which items should be evicted from the cache when it is unable to store more items. Ideally, a cache replacement policy should ensure that the queries that are not answered locally by the cache pose a minimal amount of load on the database. The load posed on the database is defined by the cost of executing all the queries, which depends on their CPU and I/O utilization. However, this information is either not available or relatively expensive to compute for the cache. In general, a cache replacement policy thus has to assume that the load of each query is equal. Therefore the goal of a cache replacement policy is to ensure that the total amount of queries that can be answered by the cache is maximized.

Most Web applications exhibit a reasonable degree of temporal locality. A high level of temporal locality indicates that many queries arriving at a certain point in time are likely to arrive again in the near future. The simplest cache replacement policy that is good at exploiting a high level of temporal locality is Least Recently Used (LRU), which stores the query results for queries that have arrived most recently. We do not investigate the effectiveness of other caching policies in this thesis.

3.2 Cache size

Besides the caching policy, the cache size is also important for the hit ratio. If the cache size is too small, a lot of potentially hot items will be evicted from the cache due to a lack of space. Hot items are those items that are frequently requested and should therefore be kept in the cache. However, the hit ratio does not increase linearly with the cache size and has an exponential dampening. This means that at a certain point, increasing the cache size will have a negligible effect on the hit ratio. Nevertheless, until this point the increase of the cache size remains relevant for the hit ratio. We therefore investigate different means to increase the cache size.

The size of a cache can be increased by two methods. The first method is distribution. In a distributed cache, each machine is storing a portion of the cache. The second method is hierarchical caching in which a hierarchy of individual caches is created. Besides increasing the cache size, both distributed and hierarchical caching also increase the throughput of the cache due to the fact that multiple machines are used.

3.2.1 Distributed caching

In distributed caching, the content of the cache is distributed among a number of physical machines. This is achieved by distributing the incoming queries between all participating machines. Different approaches to distributing the queries have been studied [16, 17]. An example of a simple approach is round-robin. In a setup with N machines, machine X will receive every Xth query. In general, this will result in a situation where a lot of query results are cached at multiple machines. This is an unwanted situation because effectively a lot of space in the cache is used up with duplicate content.

A more sophisticated approach is based on hashing. In this case, each incoming query is hashed to a number between 1 and N, representing one of the N machines. This approach results in a situation where each query result is stored at no more then one machine. There are however a number of downsides to this approach. In particular, if one machine is temporarily not available, subsequent queries will be hashed to a number between 1 and N-1. Consequently a lot of queries will be forwarded to different machines. The same issue arises with the addition of a new machine.

Consistent hashing is an alternative approach that addresses the previous problem [17]. Consistent hashing ensures that when a machine is (temporarily) added or removed, the queries that now have to be forwarded to different machines are uniformly distributed among the remaining machines. A possible performance issue of (consistent) hashing is hot spots. Hot spots are requests that are issued so often that they contribute to a significant part of the overall load posed on the cache. A hashing approach will always forward hot spot request to the same machine. Depending on the processing speed of the cache, this may become an issue.

3.2.2 Hierarchical caching

The second method of increasing the cache size is hierarchical caching. In this approach a tree of caches is created with at least two levels. Unlike in distributed caching, each machine is responsible for its own incoming query workload. If a request for a cache at level N results in a cache miss, the request is forwarded to a cache at level N+1. This process continues until there is a cache hit or until a top level cache also misses in which case the request is forwarded to the database server.

In general, the number of caches present at a high level is much smaller than the amount of caches at a low level. Caches at a high level generally have a lower hit ratio than caches at a lower level. This is due to the fact that all temporal locality of the workload has been removed by the first level (L1) caches. This phenomena is known as the filtering effect [18]. The only redundancy left in the workload perceived by the second level (L2) caches are those queries whose query results were not cached in the L1 caches due to space limitations.

3.3 A combined approach

In this section we explore the performance benefits of combining distributed and hierarchical caching. This combination is achieved by deploying distributed caches at L1 close to the edge servers and a single L2 cache close to the origin database server. In principle the L2 cache can be distributed to increase the throughput if necessary. Each L1 cache is located at a cluster of edge servers and is distributed between these edge servers using consistent hashing. If a request can not be answered by an L1 cache it is forwarded to the L2 cache.

The L1 caches are used for three reasons. First, the cache size is increased resulting in a higher hit ratio. Second, the throughput of the cache is increased because multiple machines are used. Third, the client perceived latency is decreased because a portion of queries can be answered from the local cache at the edge server's cluster, thereby avoiding WAN latency.

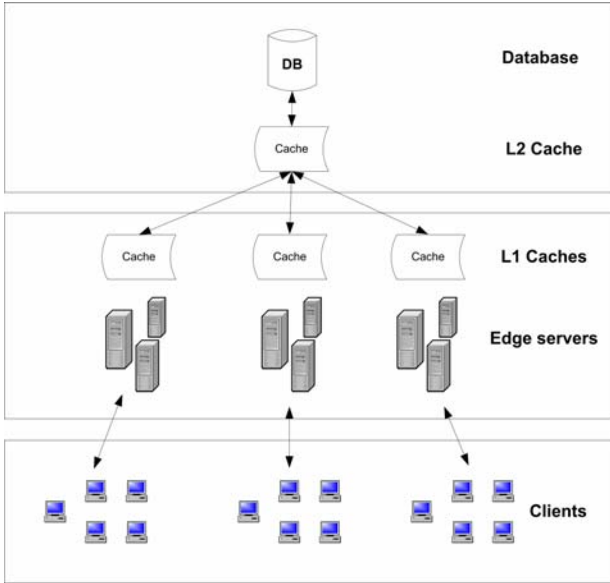

Figure 3. Query caching architecture

All edge servers that are located at the same cluster can communicate with each other over a local area network. This ensures that communication with the L1 cache is very fast and does not incur any WAN latency. However, L1 caches from different clusters do not interact with each other. As a consequence, multiple L1 cache clusters will likely store redundant data, thereby issuing identical queries to the origin database in case of a cache miss. This means that the stream of requests addressed to the origin still contains some temporal locality that can be exploited be an L2 cache cluster.

Chapter 4

Database replication

Traditional database replication systems replicate the entire database across a set of machines. The consequence of this approach is that all UDI queries have to be applied at all replicas. In general, executing a UDI query takes more time than executing a simple read query, since a UDI query always involves at least one disk write.

This observation is especially important for systems which apply a form of query caching. The reason is that only the queries that cannot be answered by the caches actually reach the database, which is always the case for UDI queries. This means that the percentage of UDI queries in the workload of the database is higher compared to a system which does not apply any form of query caching.

The performance of the replicated database can thus be increased if somehow the UDI queries do not have to be applied at all database nodes. This can be done if the data can be divided into disjoint partitions, while still ensuring that each query can be answered by at least one database node. Each partition can then be replicated on a subset of nodes. UDI queries which access data of a particular partition will then only have to be applied at that subset of nodes instead of all the nodes.

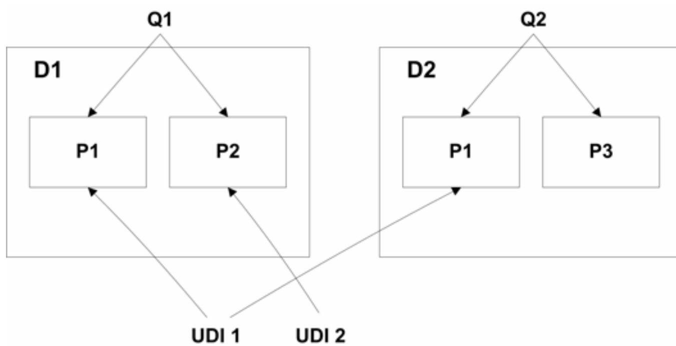

Figure 4. Example of data partitioning

Let us consider a scenario where the database is divided into three partitions {P1, P2, P3} respectively (see figure 4). Queries Q1 and Q2 respectively access data in {P1, P2} and {P1, P3}. To ensure that both queries can be executed, {P1, P2} and {P1, P3} have to be placed together on at least one node. Furthermore, each partition needs to be maintained consistent. This means that UDI queries that modify the data in the partitions will have to be applied at all nodes on which the partition is placed. In this scenario, UDI 1 modifies P1 and UDI 2 modifies P2. Partition 3 is never modified. The optimization of this placement compared against a fully replicated database is that UDI query 2 only has to be applied at database node D1 instead of all databases nodes.

A number of systems already partially replicate databases [14, 15]. In these systems the data is partitioned at the table row level. Row level data partitioning has the benefit that the placement can be very fine-grained. However, the downside of this approach is that it is very difficult to analyze whether or not all queries can be executed on at least one node for a given placement.

Here, we exploit the fact that in Web applications the query workload is based on a small number of read and write query templates. We cluster tables that are accessed together in a single query template. For example, the query template `SELECT * FROM t1, t2 WHERE t1.pk=@p1 AND t2.pk=t1.pk` accesses tables `t1` and `t2`, which will therefore be clustered together. The goal is to determine the placement of tables across the database nodes such that the resulting throughput is maximized.

4.1 Proposed algorithm

The proposed algorithm consists of four steps. First, we need to determine which tables have to be placed together to ensure that all queries can still be executed. This determines a number of (potentially overlapping) table clusters. Second, we need a method of estimating the cost of each table cluster. Third, we need of method of estimating the load of a database node given a certain placement of table clusters. Finally, we need to devise an algorithm that computes an optimal assignment for a given set of table cluster and database nodes.

4.1.1 Table cluster construction

A database consists of N tables. Read queries theoretically access between 1 and N tables. However, in general only a small number, often only one, of the N tables will be accessed by each individual query. Because in this system the applications' query workload is based on templates, they can be easily analyzed to see which tables will be accessed.

All tables that are accessed by a *read* query template together form a table cluster. Table clusters may consist of a single table. By analyzing the query templates, a list of table clusters and corresponding QID's can be created. Note that some of the table clusters can be identical. However, we choose to keep these table clusters to be able to distinguish between individual QID's in the placement algorithm. We assume that UDI queries only modify data in a single. This allows tables to be placed at arbitrary nodes as long as the *read* table clusters are placed at least at one node.

In the example depicted in Figure 5, queries Q1 (`SELECT * FROM T1, T2 WHERE T1.pk=T2.fk`), Q2 (`SELECT * FROM T2 WHERE T2.pk=@p1`) and Q3 (`SELECT * FROM T3 WHERE T3.pk=@p1`) respectively correspond the table clusters C1, C2 and C3.

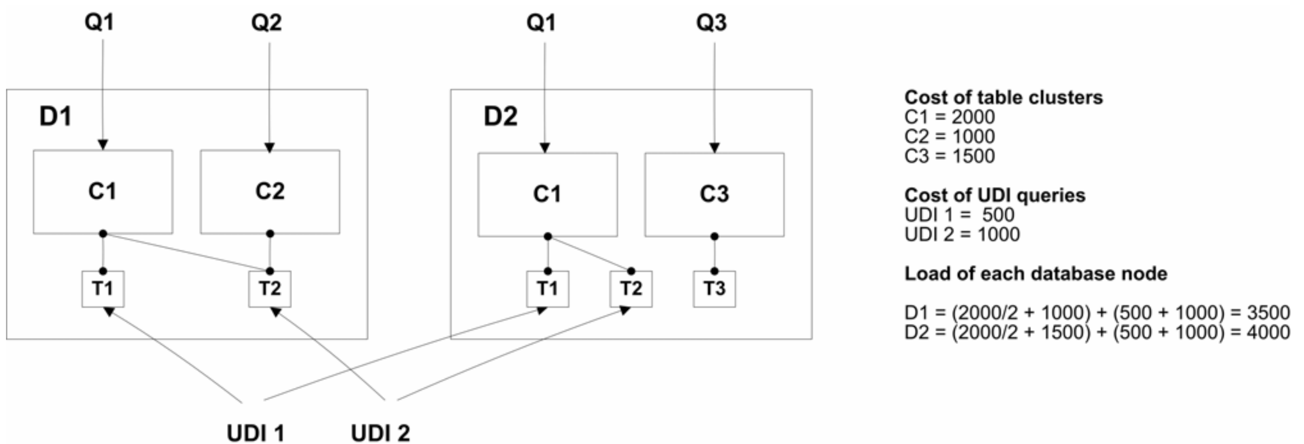

Figure 5. Example of table cluster placement

4.1.2 Table cluster cost estimation

We now need to determine the degree of replication required by each cluster based on the load it receives. To determine the load of a cluster we need to estimate the cost of the associated query template. The cost is defined by the average execution cost of the query template multiplied by its appearance in a typical

workload. The execution cost of each query template depends on the load of the node at which the query is being executed. Ideally, we would like to have an easy method of obtaining an estimated executing cost for varying loads.

Three different methods of obtaining a baseline execution cost for each query template have been investigated. The first one uses PostgreSQL's `EXPLAIN <query>` command. This command computes an estimated execution costs by analyzing the query plan of the query. The downside of this approach is that it is static and does not take the size of intermediate results of the query into account. The second method uses PostgreSQL's `EXPLAIN ANALYZE <query>` command. This command executes the query and returns the actual query execution time in milliseconds. The third method consists of measuring the response time of individual queries under a low load.

All of the different methods generate a list of query templates and the corresponding estimate of their execution time. The usefulness of these estimates depends on the correlation with actual query execution times under various loads. If this is the case then a function can be applied to the estimated query costs to obtain new estimates for various loads. Chapter 5 will evaluate the different methods and their respective correlations.

4.1.3 Database node load estimation

In a replicated database, read queries are executed at one database node, whereas UDI queries are executed at all nodes that hold the data which is modified by the UDI query. Therefore, the load of a node for a given placement is determined by two factors. The first factor is the cost incurred by the placed table clusters. The second factor is the cost of keeping the placed tables consistent. To keep the tables consistent all the UDI queries that modify the data of these tables should be executed at that node.

The load of a node is thus calculated in two phases. First, for each table cluster that is placed on the node, we check the number of nodes on which it is replicated. The cost incurred by that table cluster then becomes the cost of the table cluster divided by the number of replicas. Second, for each of the placed tables, defined by the placed table clusters, we add the cost of keeping those tables consistent. This cost is defined as the total cost of executing all UDI queries which modify data in these tables.

In the example depicted in figure 5, the three table clusters {C1, C2, C3} are assigned to two database nodes {D1, D2}. Cluster one consists of tables T1 and T2, cluster two consists of table T2 and cluster three consists of table T3. This results in the fact that node D1 is storing tables T1 and T2 and that D2 is storing tables T1, T2 and T3. To keep these tables up to date, both UDI query 1 and 2 have to be applied at D1 as well as D2. Given the costs of each table cluster and UDI query, the load of both servers for this particular placement can be calculated (see Figure 5, right side).

4.1.4 Placement algorithm

The goal of the placement algorithm is to find an optimal placement of table clusters on the available database nodes. The optimal placement is defined as the placement in which the highest load of the nodes is minimized. Two implementations of the table cluster placement algorithm are available. The first one is an exhaustive algorithm that calculates all possible placements and chooses the best one. The second one is a heuristic algorithm that only calculates a small number of possible placements and chooses the best one from within that set.

4.1.4.1 Exhaustive

The exhaustive algorithm recursively calculates all possible placements and is therefore guaranteed to find the optimal placement. However, the problem with this algorithm is that the complexity is too high. The algorithm is of order $O(2^{(N * T)} / N!)$, where T is the number of table clusters to be placed and N is the number of nodes to place them on. For a realistic number of table clusters this algorithm is too computationally expensive.

4.1.4.2 Heuristic

The heuristic algorithm calculates only a small number of possible placements (see algorithm 2). The algorithm is of order `O(N² * T)`. The algorithm starts by creating an initial placement (Step 1). This is done by ordering the clusters by decreasing cost and placing them on the nodes in a round robin fashion. This initial placement is most likely far from optimal. Remember that the optimal placement is the placement in which the maximum load of all nodes is minimized. So to improve this placement the load of the highest loaded node has to be decreased. As a secondary objective, when the load of the highest loaded node cannot be decreased any further it is still beneficial to optimize the remaining nodes.

To improve the initial placement the algorithm performs a number of optimization rounds (Step 2). Before the start of each round the maximum load of all servers is stored to see if a round improved the overall placement. On the start of each round one of the nodes is selected for optimization, initially this is the highest loaded node. After that, all nodes are ordered by decreasing load. Then the current node to be optimized (denoted as N) is selected. There are two ways for the node to be optimized. The first one is (re)moving a cluster from node N (Step 2a). The second one is replicating a cluster to a different node (Step 2b). For each cluster C placed on node N we first see if it is replicated on one of the other nodes. If this is the case then C is removed from N. If this is not the case then C is moved from N to the lowest loaded node. The intuition behind this is that moving a cluster from the highest loaded node to the lowest loaded node will improve the load balance. If this action improved the overall placement we continue with the next cluster. If this is not the case then we try to replicate C on one of the other nodes. Replication can also lower the load of a node since the cost incurred by the replicated cluster is lowered. If replication on one of the other nodes improves the overall placement we continue with the next cluster.

When the algorithm has performed step 2a and 2b for all table clusters of the currently selected node two conditions are checked (Step 3). The first condition is whether or not the current placement is an improvement compared to the placement at the start of the round (called *improvement*). The second condition is whether or not the node that was optimized in this round is the last node in the ordering (called *last node*). The algorithm then proceeds as follows. If *improvement* is not true, then the next node in the ordering will be selected for optimization. If *last node* is true but *improvement* is not true then the algorithm terminates. However, if both *last node* and *improvement* are true the optimization will start again with the highest loaded server.

```
Step 1: "Creating an initial placement"

Order clusters by decreasing cost;

For each cluster Do
       Place cluster on next node;
End For

Step 2: "Improving the initial placement"

Set done to false;
Set index to 0;
Set loadLastRound to maximum load of all nodes;

While !done Do

       Order nodes by decreasing load;

       Selected node N = nodes[ index ];

       For each cluster C placed on node N Do
```

```
            Step 2(a): "Removing a table cluster"

            If C is replicated on other nodes Then
                  Remove C from N;
            Else
                  If N != lowest loaded node Then
                        Move C from N to lowest loaded node;
                  End If
            End if

            If maximum load of all nodes decreased Then
                  Keep this placement;

                  Continue
            End If

            Step 2(b): "Replicating a table cluster"

            For each node N' excluding N Do

                  Replicate C on N';

                  If maximum load of all nodes decreased Then
                        Keep this placement;

                        Continue
                  End If

            End For

      End For

      Step 3: "Stop condition"

      Improvement = maximum load of all nodes < loadLastRound;

      If !Improvement Then
            Increment index;
      End If

      If index == number of nodes Then

            If !Improvement Then
                  done = true;
            Else
                  index = 0;
            End If

            loadLastRound = maximum load of all nodes;
      End If

End While
```

Algorithm 1. Heuristic table based data placement

The goal of the heuristic algorithm is to balance to load of all database nodes. Therefore we take the imbalance of the computed placement as the metric for the performance of the algorithm. We define the imbalance of a placement as the standard deviation of the estimated load of all nodes divided by the average load. Evaluations show that the imbalance of the placements computed by the algorithm are generally smaller than 10%.

4.2 Query routing

In a replicated database system that uses table based data partitioning, queries can no longer be sent to arbitrary database nodes. First of all, read queries can only be executed on those database nodes which store the needed tables. Second of all, UDI queries must be sent to all database nodes which store tables that are modified by the UDI query. Therefore a query router is needed that routes incoming queries to database nodes in such a way that read queries are guaranteed to execute correctly and that UDI queries are sent to the required database nodes. The query routing policy adopted by the query router can have a considerable impact on the overall performance of the system.

In this system two routing policies have been implemented. The first is a simple round robin routing policy adapted for data partitioning. The second one is a cost-based routing policy which takes the costs of executing queries into account. Both policies route UDI queries to all database nodes on which they need to be executed. However, they are discriminated by the way they route read queries.

4.2.1 Round robin per QID

Each read query that arrives at the query router is accompanied by its associated QID. Upon arrival, the set of database nodes that are able to handle this query are retrieved. All queries that have the same QID are routed to this set of nodes by means of round robin.

4.2.2 Cost-based

In cost-based routing, the cost of executing the queries is taken into account by maintaining the list of queries that are being processed by each database node. At any point during the execution of the system, each database will be busy executing N concurrent queries. Furthermore, if the maximum concurrency level has been reached there can also be another M queued queries waiting for execution. Upon arrival of a query, the set of database nodes that can handle this query are retrieved. For each node in this set we calculate its load by adding up the cost of the N + M pending queries (see algorithm 2). The cost of each query is equal to the estimated cost of the associated query template. Finally we select the database node with lowest load to execute the query.

```
Select all nodes that can handle this request;

For each node Do

       Get all pending and queued requests;
       Set cost for this node to zero;

       For each request Do
              Add estimated cost for this request's QID to cost;
       End For

End For

Select the node with the lowest cost;
```

Algorithm 2. Cost-based routing

Chapter 5

Evaluation

In this chapter a number of different parts of the work done for this thesis will be evaluated. First, the query caches will be evaluated. Second, the query cost estimation methods will be evaluated. Third, the novel database replication technique as well as the cost-based query routing policy will be evaluated. Finally the performance of the combination of query caching and database replication will be evaluated.

5.1 Experimental setup

All the experiments done for this thesis are based on the RUBBoS benchmark Web application [23]. RUBBoS is a benchmark that models a news website like http://slashdot.org. All experiments have been performed on the DAS2 cluster of the Vrije Universiteit (VU). The DAS2 cluster consists of 72 homogenous nodes. Each node has two 1Ghz. Pentium III processors, 2GB of RAM, 20GB hard disk and are interconnected via a fast Ethernet network.

The RUBBoS database consists of 8 tables, which are used to store information of users, stories and comments. The query workload of the RUBBoS benchmark consists of 36 query templates: 25 read query templates and 9 UDI query templates. The 25 read query templates result in 25 table clusters of which 9 are unique.

We adapted the RUBBoS application such that each query issued from the application is logged to a file along with the corresponding QID and the actual parameters. The log files act as input data for the various experiments and are replayed to test the performance of various configurations under reproducible circumstances.

The query logs for the RUBBoS benchmark are created with the help of the Emulated Browser (EB) program that is packaged in the distribution. This program simulates a number of clients concurrently visiting the RUBBoS Web site. The number of concurrent emulated clients as well as the transition matrix to be used can be set in the configuration file. The transition matrix describes the chance that a client will click from page X to page Y. A number of transition matrices are shipped with the distribution resulting in different workloads. The chosen transition matrix determines whether the workload will contain UDI queries, search queries or both. This is important because the costs of UDI and search queries are a couple of orders of magnitude larger than the cost of other (read) queries. In the following experiments two kinds of workloads have been used. The first one contains both UDI queries and search queries and is called the *search* workload. The second one contains UDI queries but does not contain search queries. This workload is called the *nosearch* workload.

The RUBBoS Web application logs each request (a query, its QID and the actual parameters) to a file. Each run of the emulated browser program results in a single log file. The maximum number of concurrently simulated clients for each run is 30. A higher number would give irrelevant results because the load on the application and database server would be too high. However, for serious testing of the various parts of the system a heavier workload is necessary. To achieve this, the EB program is run multiple times and the resulting log files are merged together into a big log file to create workloads up to any necessary intensity. The log files are merged rather than appended to maintain the original temporal ordering of the individual

log files. The big log file's workload is therefore equivalent to creating this log file in one run with a big number of emulated clients. Every run lasts for two hours to create a trace of reasonable length which gives both caches and database servers time to warm up.

5.2 Query caching

The performance of query caching was evaluated using trace driven simulations. In these experiments, the query request rate is controlled by varying the number of emulated browsers used to obtain the query log. The output stream of each L1 cache can optionally be merged together and fed into a simulated L2 cache to evaluate the performance of hierarchical caching. Both the L1 and L2 caches use LRU as the cache replacement policy.

In the first experiment we analyze the impact of the cache size on the cache hit ratio in distributed caching. In this case the average cache hit ratio of ten individual L1 caches is measured. The cache size is expressed in the number of entries that the each L1 cache is allowed to store. Given an average size of the query results the corresponding total memory needed can be easily calculated. We could also have evaluated the hit ratio of L1 caches which apply consistent hashing, but this is effectively the same as if there would be a single L1 cache with more memory.

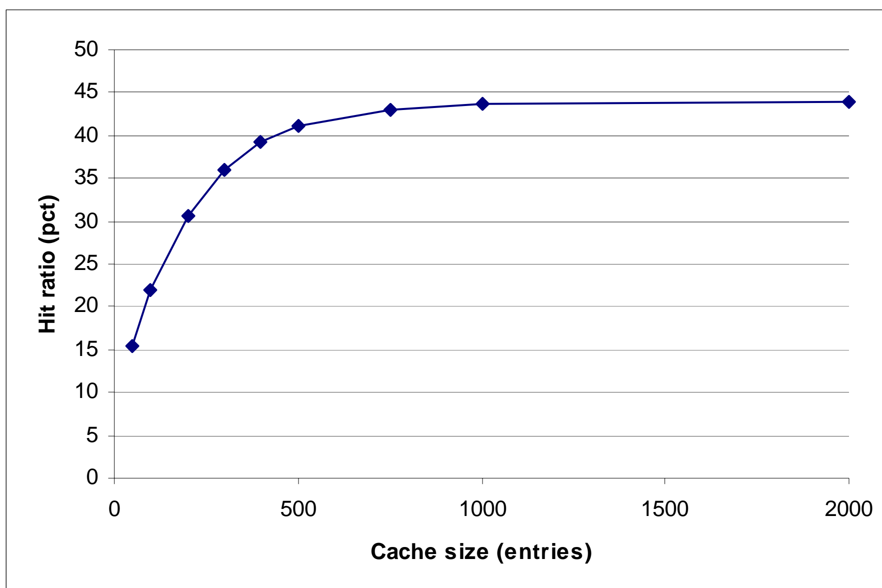

Figure 6. L1 average hit ratio (415q/s search workload)

As can be seen in figure 6, the L1 caches can have a significant hit ratio of up to 44%. The reason that the cache performs so well is that the RUBBoS application has a high query locality that is easily exploited by the LRU cache replacement policy.

In the second experiment the cache hit ratio of a single L2 cache is investigated (see Figure 7). The cache size for each of the ten L1 caches is fixed at 2000 entries. This ensures that the L1 caches have a maximum cache hit ratio so that only the effect of the L2 cache size is measured. Queries that can not be answered by an L1 cache are forwarded to the L2 cache.

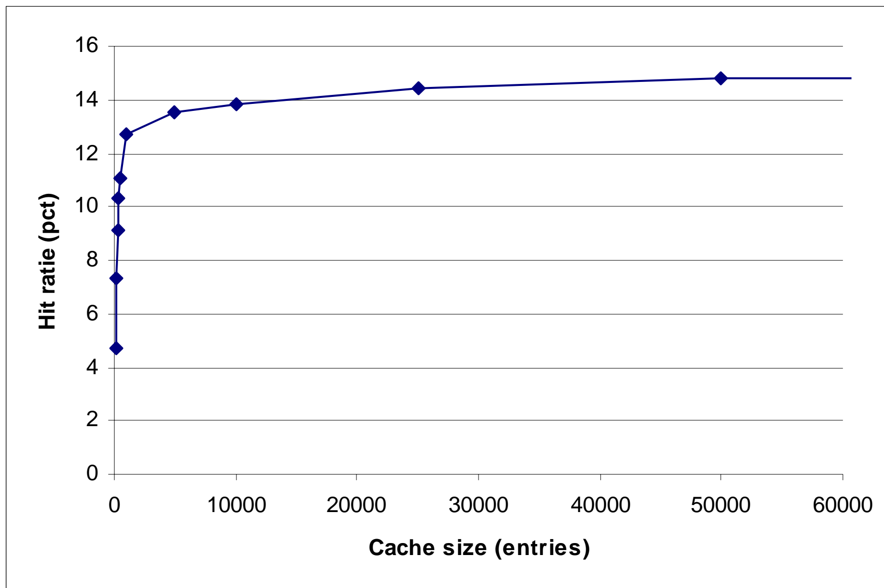

Figure 7. L2 hit ratio (L1 cache size: 2000, 415q/s search workload)

As shown in figure 7, the L2 cache hit ratio can get up to 15% as long as the cache size is big enough. Although this is not nearly as good as the hit ratio of the L1 caches, a 15% hit ratio does validate the usefulness of adding an L2 cache. We also note that even a relatively very small L2 caches can have a significant impact in terms of the reduced workload addressed to the database.

In the third experiment we investigate the impact of the request rate on the cache hit ratio of both the L1 and L2 caches. The request rate is controlled by varying the number of emulated browsers used for collecting the query logs. The cache size for the L1 and L2 caches is fixed at respectively 2000 and 100000 items.

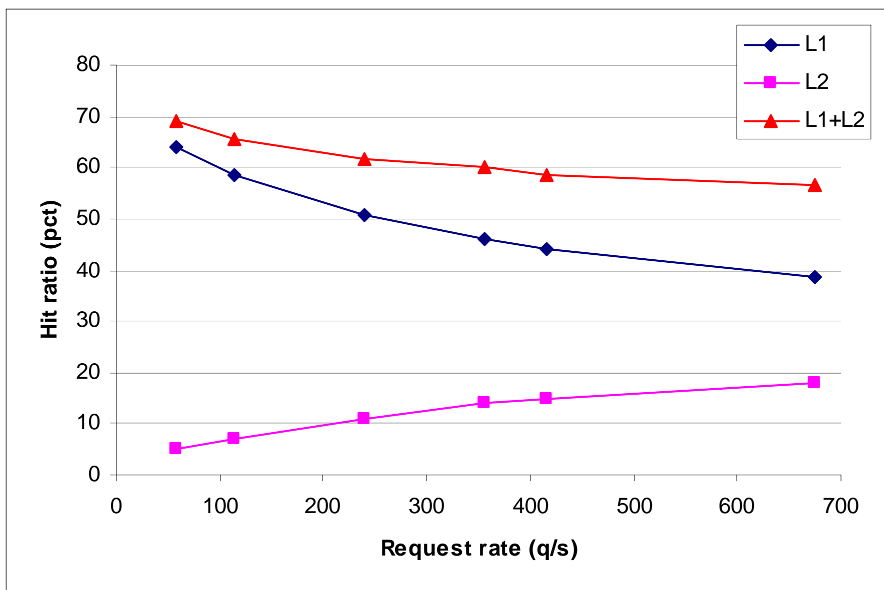

Figure 8. Cache hit ratio for varying request rates (Cache size L1:2000, L2:100000, search workload)

As figure 8 shows, an increasing query request rate has two consequences. The first is that the average cache hit ratio of the L1 caches decreases. The second is that the cache hit ratio of the L2 cache increases. However, for both the L1 and L2 cache this effect stabilizes as the request rate increases.

The decrease of the average L1 cache hit ratio can be explained by the increase of the number of UDI queries. As the request rate increases, so does the rate at which UDI queries are issued. Each UDI query modifies data in the database which in turn triggers invalidations of query results stored in each L1 cache. The number of query results that are invalidated in a cache have a negative impact on its hit ratio.

The increase of the L2 cache hit ratio is a result of the increased request rate. As the request rate is increased we add more L1 caches such that the load posed on each L1 cache is balanced for each test result. The queries forwarded from each L1 cache to the L2 cache will have some overlap. This redundancy in the incoming workload of the L2 cache is the only reason it has a reasonable hit ratio. Increasing the number of L1 caches increases the redundancy, which in turn increases the hit ratio of the L2 cache.

5.3 Query cost estimation

In chapter three different methods of obtaining an estimated query response time have been described. This experiment evaluates the correlation between the estimated query response time and the actual query response time of each query template. It is important that the estimated query response times correlate with the actual response times, especially for frequently accessed query templates. This is because the total error margin of the cost estimation is higher for frequently accessed queries. A cost estimation method thus produces the best results if the correlation for those important query templates is strong.

Out of the three different methods, the first method was based on PostgreSQL's `EXPLAIN` command. Figure 9 shows the correlation between the estimated and actual response times of this method. Note that the scale of both axes is logarithmic.

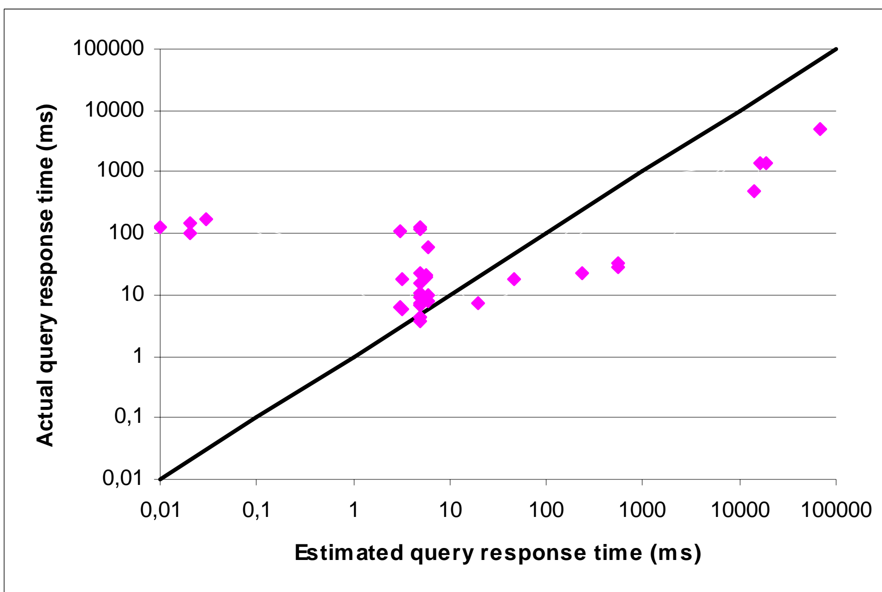

Figure 9. Correlation between response times using the `EXPLAIN` method

The correlation of the `EXPLAIN ANALYZE` method is shown in figure 10. The third method was based on executing a large number of queries under low load. This results of this method are shown in figure 11.

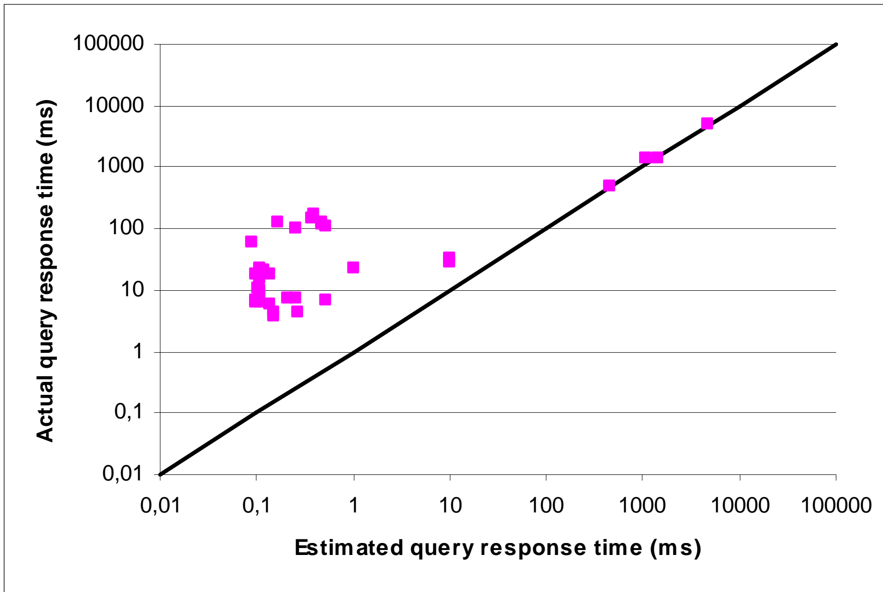

Figure 10. Correlation between response times using the `EXPLAIN ANALYZE` *method.*

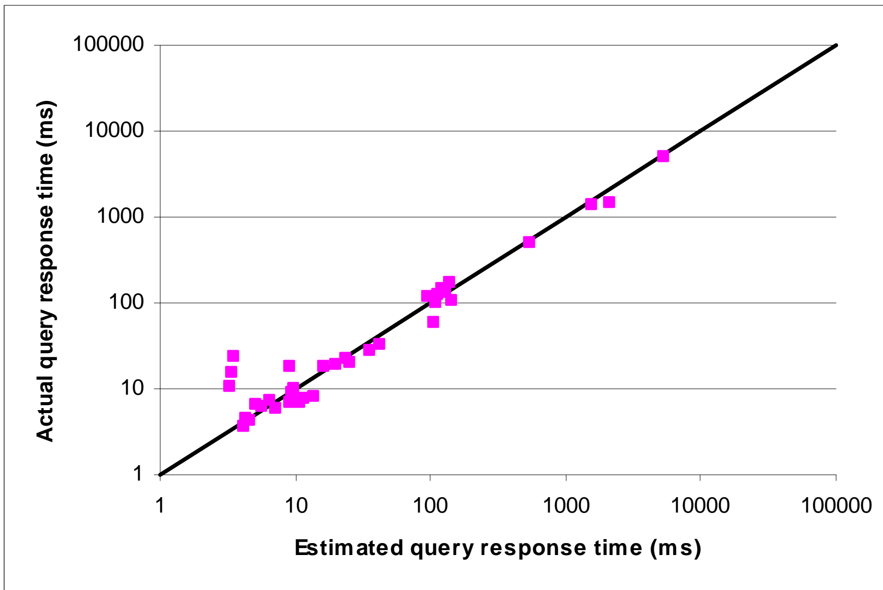

Figure 11. Correlation between response times of the third method

As seen in Figure 9, the correlation of method one is very weak. The actual query response times for a similar estimated response time are sometimes multiple orders of magnitude apart. However, the correlation for the other two methods is stronger. The third method clearly shows the highest correlation between the estimated and the actual query response times. This is due to the way the estimates are computed in this method. Method three measures the response time under low load, thereby avoiding any queuing latency. This also results in the fact that query templates that contribute to a significant part of the workload will appear more often and thus result in a more accurate average.

5.4 Database replication

In this section, we compare the performance of our novel data placement technique (denoted as *partitioned*) with a fully replicated placement (denoted as *replicated*). In the experiments we used 4 servers that each run

a database node and an additional server that runs the query router.

As noted earlier, the query router can impact the performance. Therefore we studied the effect of three different policies on the performance. The first one is Round Robin (RR), which can only be used if the database is fully replicated. The second one is Round Robin per QID (RRQID), which is the equivalent of RR for a partitioned database. The third one is Cost-Based (CB), which can only be used in conjunction with a partitioned database. The goal of a replicated database is to minimize the overall query latency. Therefore, the measured performance metric is the query response time.

To create the log files necessary for testing the replicated database, the query cache simulator outputs two log files. The first one contains the requests from the input log that were missed by the L1 cache. The second one contains the requests from the input log that were missed by the L1 and the L2 cache. These two logs combined with the original input log can be used as input for the replicated database to see performance under different circumstances.

The first experiment evaluates the performance of different combinations of routing policies and replication techniques for a given workload. The experiment is done for both the *search* and *nosearch* workload.

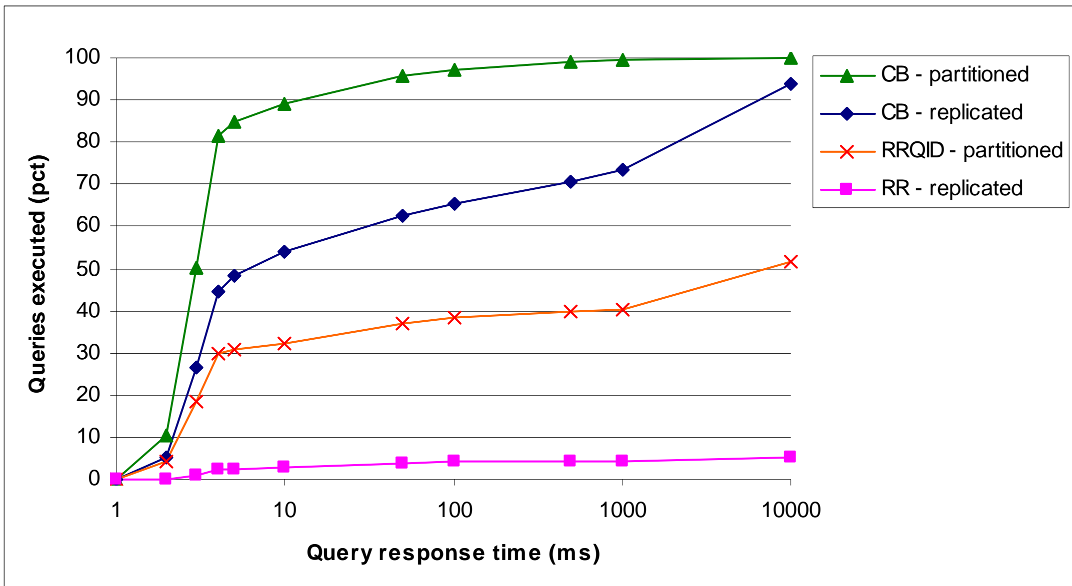

Figure 12. Cumulative query response time (415q/s search workload)

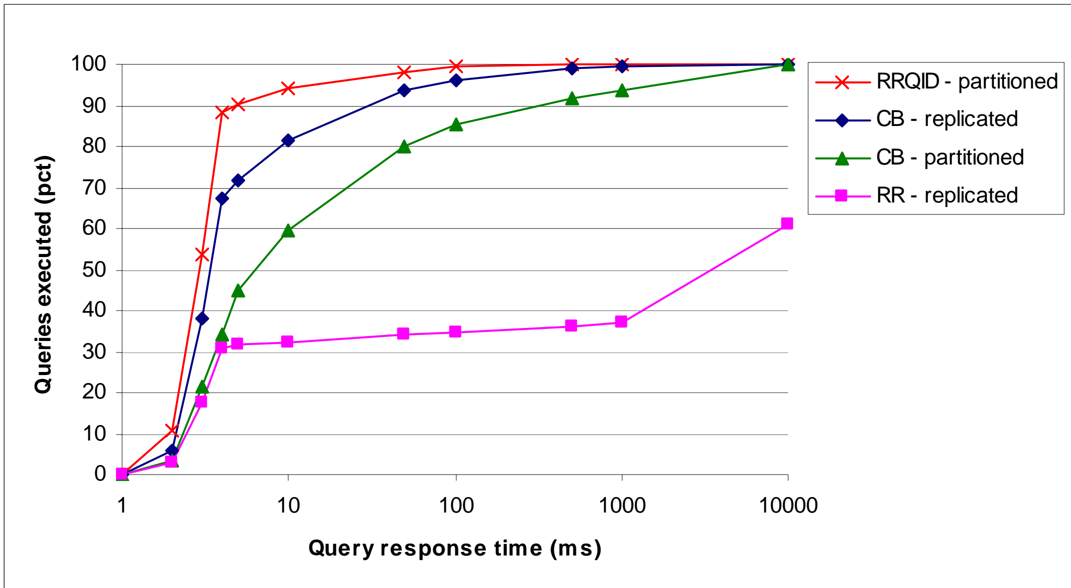

Figure 13. Cumulative query response time (415 q/s nosearch workload)

The performance of the *search* workload is shown in Figure 12. We first see that the routing policy has a greater impact on the performance than the method of replication. Second, data partitioning outperforms full replication when equivalent routing policies are used (RR and RRQID). The best combination is cost-based routing with data partitioning which is able to execute more than 95% of all queries within 100ms, whereas traditional full replication with round robin routing can only execute 5% of all queries within 100ms.

The reason that cost-based routing performs so well in this experiment is due to the fact that the estimated execution time of each query in this workload ranges from 3.7ms to 4930ms. Where the high query response times originate from search queries. Because there is so much difference in the execution time, cost-based routing will be able to balance the workload posed on each database node, whereas other approaches such as RR and RRQID can not.

The similar experiment done with the *nosearch* workload shows two major differences with the previous experiment (see figure 13). First, RRQID with data partitioning is the best performing combination. Second, cost-based routing performs better in combination with full replication than with data partitioning.

The reason that not cost-based but round robin per QID routing comes out on top is most likely due to the fact that the estimated cost for each query still has a relatively high error margin. Without the search queries the cost of each query ranges from 3.7ms to 147ms, with more than 85% of all queries under 25ms. This indicates two potential problems that effect cost-based routing. The first one is that all queries cost more or less the same, making routing based on cost less relevant. The second one is that the error margin for low cost queries has a relatively large impact. This results in the fact the cost-based routing might not estimate the load of each database node correctly, thereby routing queries to the wrong nodes.

The fact that CB routing performs poorer than RRQID is also a possible explanation why cost-based routing performs better with full replication instead of data partitioning. In data partitioning, each read query can only be routed to a subset of nodes. An imbalance in the cost-based routing algorithm thus chooses a possibly wrong node from a smaller set of nodes which has a greater impact on the overall imbalance.

5.5 Combined approach

In this experiment, we evaluate the performance of a combined approach. Five different configurations are used to evaluate their relative performance. The baseline configuration is a traditional replicated database using round robin routing with no form of query caching (RR – replicated). That configuration is then changed by first adding a distributed cache (L1) and then adding a hierarchical cache (L2). After that the

configuration is changed to use data partitioning both with RRQID and CB routing.

In this experiment a heavier query workload (768 q/s) is used than in the previous experiments. This is because the query caches will filter out up to 60% of the original workload. If this is done for the previously used workload (415 q/s) the remaining workload for the database would be too easy and all approaches would perform exactly the same.

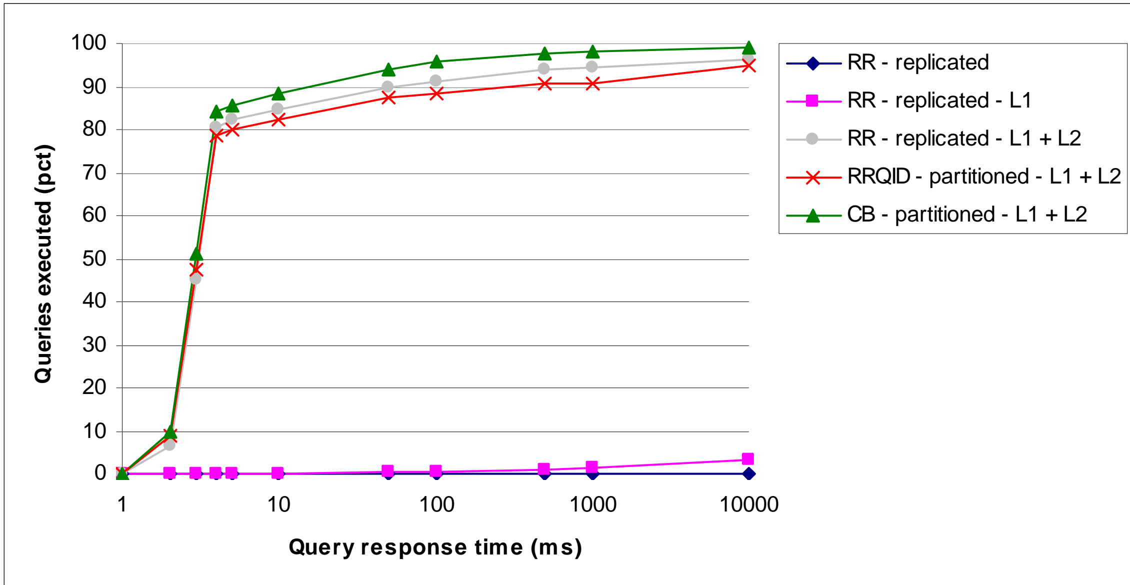

Figure 14. Cumulative query response time (768 q/s search workload)

As can be seen from figure 14, the traditional fully replicated database with or without L1 caches are unable to handle this workload. However, the addition of the L2 cache ensures that even a simple fully replicated database can handle the workload. Furthermore, the data partitioning in combination with cost-based routing performs best for this specific workload. An unexpected result is that the RRQID routing together with data partitioning and hierarchical caching performs slightly worse than the equivalent configuration with a fully replicated database. However, due to the fact that the difference between the different methods is so small we can not draw any conclusions from it.

Chapter 6

Conclusion

This thesis has introduced three approaches to increase the scalability of data-intensive Web applications. Existing solutions are based on a single approach which only focus on one aspect of the scalability bottleneck. In contrast, this thesis combines all three different approaches in order to maximize the scalability.

We first studied the impact of distributed and hierarchical query cache organization strategies on the workload addressed to the database. We proposed a combined approach in which we have two levels of caching. At each cluster of edge servers, an L1 cache is placed which is distributed between the edge servers. Furthermore we placed an L2 cache close to the database server. The L1 caches remove most of the temporal locality in the workload. However, the L2 cache is still able to get a reasonable hit ratio due to the fact that multiple L1 caches will send identical queries to the L2 cache after a part of their cache has been invalidated due to a UDI query. Our evaluations show that the L1 and L2 caches can get hit ratios of up to 44% and 15% respectively, which combined leads to a hit ratio close to 60%.

While query caching is a useful technique to decrease the number of read queries in the workload of the database, all UDI queries still have to be processed by the database. Because UDI queries are more expensive than simple read queries they can become a serious performance bottleneck. Traditional database replication techniques replicate all the data on all the database nodes. This results in the fact that all UDI queries have to be executed at all nodes, which can result in poor performance. This thesis introduced a novel data replication technique which applies partial data replication at the table level in order to decrease to total number of executed UDI queries. We were able to apply partial replication by taking advantage of the fact that a Web application's query workload is based on a small set of read and write query templates. These templates can be easily analyzed to see which tables will be accessed by each template. Our evaluations show that our replication technique can increase the database throughput from 200% up to 700% compared to a fully replicated database.

In a partially replicated database, queries can no longer be sent to arbitrary database nodes. Read queries can only be executed on nodes which store the needed data and UDI queries must be executed on all nodes which store the data that they modify. A query router takes care of these issues. We showed that the adopted query routing policy has a significant impact on the overall performance of the database. This thesis introduced a cost-based routing policy which takes the cost of queries that are being processed by each database node into account. Our evaluations show that for workloads in which the individual cost of each query is very different, cost-based routing can increase the database throughput up to 200% compared to traditional round-robin routing.

Finally, this thesis evaluated the performance of the simultaneous use of these three approaches. The setup consisted of a CDN that used L1 and L2 caches, a database cluster at the origin site with data replicated according to our novel data replication algorithm and a cost-based routing policy. Our evaluations show that, for a fixed workload, this setup can process 95% of all queries within 100ms compared to 1% for a fully replicated database without any form of query caching using a round-robin routing policy.

The simultaneous use of three different approaches have shown to significantly increase the scalability of

Web applications in CDN environments. We therefore hope that the work done in this thesis will be adopted by CDNs in order to increase the scalability of current and future Web applications, thereby improving the World Wide Web experience for everyone. Future research could further increase the scalability by improving the cost-based routing policy such that it also performs well for situations where the costs of individual queries are not very different.

Acknowledgments

The finishing of this master thesis ends the five year period that I have studied at the Vrije Universteit in Amsterdam, three years for my bachelor degree and two years for my master degree. Even though I started studying computer science only because all the other studies did not interest me at all, I am very satisfied with the way things turned out. Looking back I have learned a lot of interesting things from a lot of interesting people.

For this master thesis specifically, I would like to thank a few people who all assisted me in different ways. First of all, I would like to thank my supervisors Guillaume Pierre and Swaminathan Sivasubramanian. Thank you for your guidance and constructive criticism both on technical issues and the actual writing of the thesis. Next, I would like to thank my parents for showing interest in the progress of my thesis even though it was most likely very difficult to comprehend. Furthermore, I would like to thank my colleagues Yves Groen and Tom Verhagen for understanding that I could not work full time with them just yet. Last but not least, I would like to thank my girlfriend Diana Veerman for being there for me, for listening to my ideas and problems and for providing useful suggestions for the content of this thesis.

Bibliography

[1] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Scalable Database Replication Middleware. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, 2002.

[2] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are Quorums an Alternative for Data Replication. *ACM Transactions on Databases*, Sept. 2003.

[3] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proceedings of the International Conference on Very Large Databases*, 2000.

[4] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proceedings of the 19th International Conference on Data Engineering*, March 2003.

[5] C. Amza, G. Soundararajan and E. Cecchet. Transparent Caching with Strong Consistency in Dynamic Content Web Sites. In *Proceedings of the 19th annual international conference on Supercomputing*, 2005.

[6] C. Bornhvd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, June 2004.

[7] P. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL server. *Data Engineering*, June 2004.

[8] S. Sivasubramanian, G. Pierre, M. van Steen and G. Alonso. GlobeCBC: Content-blind Result Caching for Dynamic Web Applications. *Technical report IR-CS-022*, Vrije Universiteit, June 2006.

[9] M. Ronstrom and L. Thalmann. MySQL Cluster Architecture Overview. *MySQL Technical White Paper*, April 2004.

[10] C. Plattner and G. Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM International Middleware Conference*, 2004.

[11] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of SIGMOD International Conference on Management of Data*, 2005.

[12] C. Amza. Conflict-Aware Replication for Dynamic Content Web Sites. *PhD thesis*. Rice University, Houston, Texas, 2002.

[13] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. In *Proceedings of the EuroSys Conference*, 2006.

[14] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: autonomic data replication for web applications. In *Proceedings of the 14th international conference on World Wide Web*, pages 33-42, 2005.

[15] J. M. Milan-Franco, R. Jimenez-Peris, M. Patio-Martnez, and B. Kemme. Adaptive

middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*, October 2004.

[16] G. Soundararajan, C. Amza, A. Goel. Database replication policies for dynamic content applications. In *Proceedings of the EuroSys Conference*, 2006.

[17] D. Karger, A. Sherman, A. Berkhemier, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *Proceedings of the Eighth International World Wide Web Conference*, May 1999.

[18] Carey Williamson. On filter effects in web caching hierarchies. *ACM Transactions on Internet Technology*, Feb 2002.

[19] J. Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *Proceedings of the IEEE International Conference on Autonomic Computing,* June 2006

[20] J. Challenger, P. Dantzig, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, Nov 2004.

[21] W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, and D. Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proceedings of the Twelfth international conference n World Wide Web*, 2003.

[22] Akamai, http://www.akamai.com.

[23] RUBBoS benchmark web application. http://jmob.objectweb.org/rubbos.html