Autonomic Data Placement Strategies for Update-intensive Web applications

Swaminathan Sivasubramanian   Guillaume Pierre   Maarten van Steen
Dept. of Computer Science, Vrije Universiteit
Amsterdam, The Netherlands
Email:{swami,gpierre,steen}@cs.vu.nl

Abstract

Edge computing infrastructures have become the leading platform for hosting Web applications. One of the key challenges in these infrastructures is the replication of application data. In our earlier research, we presented GlobeDB, a middleware for edge computing infrastructures that performs autonomic replication of application data. In this paper, we study the problem of data unit placement for update-intensive Web applications in the context of GlobeDB. Our hypothesis is that there exists a continuous spectrum of placement choices between complete partitioning of sets of data units across edge servers and full replication of data units to all servers. We propose and evaluate different families of heuristics for this problem of replica placement. As we show in our experiments, a heuristic that takes into account both the individual characteristics of data units and the overall system load performs best.

1. Introduction

Edge service architectures have become the most widespread platform for distributing Web content over the Internet. Commercial Content Delivery Networks (CDNs) like Akamai [1] and Speedera [14] deploy edge servers around the Internet that locally cache (static) Web pages and deliver them from servers located close to the clients. However, the past few years have seen significant growth in the amount of Web content generated dynamically using Web applications. These Web applications are usually database driven and generate Web content based on individual user profiles, request parameters, etc. When a request arrives, the application code examines the request, issues the necessary read or update transactions to the database, retrieves the data and composes the page which is then sent back to the client.

Traditional CDNs use techniques such as fragment caching whereby the static fragments (and sometimes also certain dynamic parts) of a page are cached at the edge servers [5, 10, 6]. However, the growing need for personalization of content (which leads to poor temporal locality among requests) and the presence of data updates significantly reduce the effectiveness of these solutions.

To handle such applications, CDNs often employ edge computing infrastructures where the application code is replicated at all edge servers. Database accesses become then the major performance bottleneck. This warrants the use of database caching solutions, which cache certain parts of the database at edge servers and are kept consistent with the central database. However, these infrastructures require the database administrator to define manually which part of the database should be placed at which edge server.

In our earlier work, we described the design and implementation of GlobeDB, an autonomic replication middleware for Edge Computing infrastructures. The distinct feature of GlobeDB is that it performs *autonomic* placement of application data by monitoring the access to the underlying data. Instead of replicating all data units at all edge servers, GlobeDB automatically replicates the data only to the edge servers that access them often. GlobeDB provides Web-based data-intensive applications the same advantages that CDNs offer to traditional Web sites: low latency and reduced network usage [13].

The data placement heuristics developed in this previous work assumed that the number of data update requests is relatively low compared to that of the read requests. While this assumption is often true, there exists a class of applications that receive a large number of updates. For example, a stock exchange Web site which allows its customer to bid or sell stocks in real time is likely to receive large quantities of updates (the New York Stock Exchange receives in the order of 700 update requests per second [8]).

Replicating an update-intensive application while maintaining consistency among the replicas is difficult because each update to a given data unit must be applied at every server that holds a copy of it. In such settings, creating extra replicas of a data unit can have the paradoxical effect of increasing the global system's load rather than decrease it. This may be a significant problem as the service time to update a data unit is usually an order of magnitude higher than that to read a data unit.

Placing replicas for update-intensive applications warrant algorithms that take the client access patterns as well as the system load into account. Our hypothesis is that there is a continuous spectrum of choice in placement configurations between complete partitioning of sets of data units (across edge servers) and full replication of data units to all servers. The best performing configuration in this spectrum depends on the system load and individual access characteristics of the data. We believe manually placing data will generally lead to poor performance. In this paper we propose algorithms that automatically take the correct placement decision.

The contributions of this paper are twofold. First, we propose several heuristics for the data placement problem and identify the best performing heuristic based on extensive simulations. Second, we identify the open issues that need to be addressed in realizing the autonomic CDN we envisage here. None of these contributions were reported in our earlier work [13].

The rest of this paper is structured as follows: Section 2 presents our system and application model and the architecture of GlobeDB. Section 3 discusses several issues concerning the problem of optimal data placement and defines the problem formally. Section 4 presents heuristics to address this problem. Section 5 presents the relative performance of heuristics using experimental evaluations. Section 6 discusses the related work and Section 7 presents several open issues. Finally, Section 8 concludes the paper and discusses future work.

2. System Model

In this section, we present our application model and a brief overview of GlobeDB and its architecture.

2.1. Application Model

A Web application is made of code and data. The code is usually written using standard dynamic Web page technology such as PHP and is hosted by each edge server. It is executed each time the Web server receives an HTTP request from its clients, and issues read/write accesses to the relevant data in the database to generate a response.

We can classify the database queries used by applications based on the number of rows matched by the query selection criterion. We refer to the queries based on primary keys of a table or that result in an exact match for only one record as *simple queries*. An example of a simple query is “Find the customer record whose userid is ‘xyz’.” Queries based on secondary keys and queries spanning multiple tables are referred to as *complex queries*. An example of a complex query is “Find all customer records whose location is ‘Amsterdam’.”

Web applications can be classified into two types based on the nature of the queries used by them. The first type are applications whose query workload tend to consist of mostly reads and a significant portion of the workload consists of complex queries. In these applications, updates are

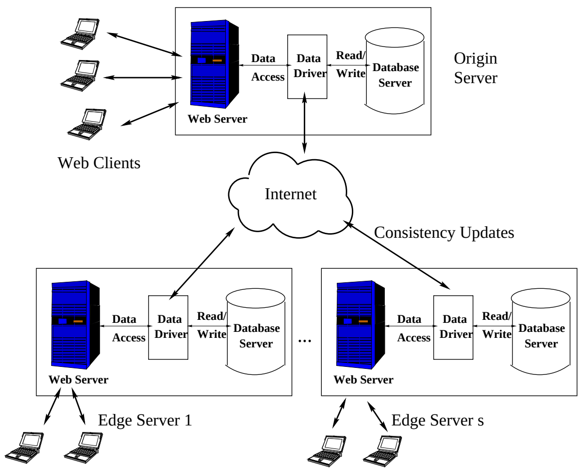


Figure 1. System Architecture - Edge servers serving clients close to them and interactions among edge servers goes through Wide-area network.

assumed to be rare and are usually not in the critical path of serving user requests, i.e., they need not be reflected to the user immediately and their answers need not be accurate. Examples of these applications include E-bookstores such as Amazon.com or Ebay.

The other class of applications are data-intensive applications. Their data workload tend to consist of a majority of simple queries and only a few complex queries. Furthermore, these applications also have a significant number of updates and these updates have to be reflected upon the generated Web content immediately. An example of this type of application is an electronic stock exchange. In this paper, we focus on the latter class of applications as it is the most demanding. However, our results can also be applied to the first class of applications.

2.2. GlobeDB: Overview and Architecture

GlobeDB is an autonomic data replication middleware system for edge computing infrastructures. GlobeDB replicates the application code and its underlying data to edge servers so that Web documents can be generated locally, resulting in reduced client latency. Instead of replicating all application data at all edge servers, GlobeDB automatically replicates them only to edge servers which access them often. GlobeDB uses a *transparent* replication model. Therefore, the application developer need not worry about replication issues and can just stick to the functional aspects of the application.

The architecture of GlobeDB is illustrated in Figure 1. An application is assumed to be hosted collectively by a fixed set of s edge servers spread across the Internet. Communication between edge servers is realized through wide-area networks incurring wide-area latency. Each client is assumed to be redirected to its closest edge server using enhanced DNS-based redirection [7]. Furthermore, for each session, a client is assumed to be served by only one Web server.

We assume that the database is split into n data units, D_1, D_2,$\cdots$, D_n, where a data unit is the granule of replication. Each unit is assumed to have a unique identifier, which is used by the data driver to track it. An example of a data unit is a database record identified by its primary key. We discuss more about the choices of data granularity in Section 7.

Each edge server runs a Web server, an application server, a database server and a data driver. The data driver is the central component of our system. It is responsible for finding the relevant data for the code either locally or from a remote edge server. Additionally, when handling write data accesses, the driver is also responsible for ensuring consistency with other replicas.

GlobeDB enforces consistency among replicated data units using a simple master-slave protocol: each data unit has one master server responsible for serializing concurrent updates emerging from different replicas. GlobeDB assumes that each database transaction is composed of a single query which modifies at most a single data unit. When a server holding a replica of a data unit receives a read request, it is answered locally. If the server does not have a replica, then the request is forwarded to the closest edge server that holds a replica. When a server receives an update request, it forwards the request to the master of the data unit, which processes the update request and propagates the result to the replicas. Note that this is sometimes called eventual consistency.

To perform autonomic replication, each application is assigned one *origin server*, which is responsible for making all application-wide decisions such as placing data units at edge servers. The origin server selects replica placement periodically to handle changes in data access patterns and the creation of new data units. Throughout this paper, we assume that most of requests result only in simple read or write queries. Web requests leading to complex queries are assumed to be served by the origin server.

3. Replica placement

Placing replicas to minimize access latency is the central problem studied in this paper. In this section, we first discuss different metrics that influence replica placement. We then formulate the problem mathematically and show that finding the optimal placement is NP-complete.

3.1. Placement Issues

The goal of replica placement is to select servers where each data unit should be placed such that the total request latency is minimized. The total latency of a request can be defined as the sum of network latencies incurred during the treatment of the request, and the internal server latency to execute the necessary database operations. The internal latency highly depends on the load the server experiences at the time of the request and on the nature of the operation (read or write). There are several factors that influence the total latency and thus the choice of an optimal placement: read-write ratio of data, system load, bandwidth consumed for maintaining consistency, client distribution, etc.

For instance, if the workload contains very few updates, then the system can improve client latency by replicating the data around the Internet. However, if the data units receive a large number of updates and only a few reads, increasing the replication degree can degrade performance because each replica has to process updates issued not only by its clients but also by clients accessing other edge servers. It is then usually better to move the data to the edge server where most of the updates are originated and make it its master.

Another metric closely related to the number of updates is the system load. If the system is underloaded, data units can be replicated in many places to improve client latency. However, under heavy loads, creating more replicas can increase the global system load depending on the data read-write ratio. In such heavy loads, reducing the replication degree can decrease the server's load and therefore improve the internal latency. In extreme scenarios, it is often best to move each data unit to the edge server that generates most updates, leading to a complete partitioning of data across edge servers.

It can be seen that the system must choose a point in the continuous spectrum between data partitioning and full replication based on several factors such as read-write ratio, system load etc. We believe performing this decision manually will generally lead to poor performance and propose algorithms that aid the system to perform this decision automatically.

3.2. Problem definition

Let r_{ij} and w_{ij} denote the number of read and write accesses made by clients of edge server E_j to data unit D_i. We define the following variables:

$$x_{ij} = \begin{cases} 1, \ if\ D_i\ is\ stored\ at\ E_j \\ 0, \ otherwise \end{cases} \tag{1}$$

$$m_{ij} = \begin{cases} 1, \ if\ E_j\ is\ the\ master\ of\ D_i \\ 0, \ otherwise \end{cases} \tag{2}$$

The goal of replica placement is to find the placement configuration x and master configuration m such that the client latency, $lat(x, m)$, is minimized. We define lat as follows:

$$lat(x, m) = \sum_{i=1}^{n} lat_i(x, m)$$

where lat_i is the sum of total latencies incurred by the client requests accessing D_i. lat_i comprises the network latency incurred during the treatment of the request and the internal server latency to execute the relevant database operations. It is defined as:

$$lat_i(x,m) = NW\_latency_i(x,m) + Intlat_i(x,m)$$

$Nlat_i$ is the sum of network latencies incurred by client requests to access D_i. Usually a client is redirected to its closest edge server which executes the query on its local database or at a remote edge server if it does not have a local replica. Therefore, $Nlat_i$ comprises the latency from a client to its closest edge server, E_j, and latency from E_j to the server that can serve it. In case of a read request, this is the closest server that holds a replica of D_i. For a write request, it is the master, E_{M_i}. Note that client-to-server latency is unavoidable in any case and independent of the placement of data units. We treat it as a constant, and in the rest of the paper aim to minimize only the latency between edge servers. We thus have:

$$Nlat_i(x,m) = \sum_{j=1}^{s}(r_{ij} * lat(E_j, E_{nl_i(j)})) + \sum_{j=1}^{s}(w_{ij} * lat(E_j, E_{M_i}))$$

where $E_{nl_i(j)}$ is the closest server to E_j holding a replica of D_i and $lat(E_j, E_{nl_i(j)})$ is the network latency between them. Similarly, $lat(E_j, E_{M_i})$ is the network latency between E_j and D_i's master (E_{M_i}).

$Intlat_i$ is the sum of internal server delays incurred by client requests at edge servers performing the necessary operations. For read requests, this corresponds to delays incurred at the edge server, $E_{nl_i(j)}$, that will execute the query in its local database. For write requests, this corresponds to delays incurred at the master server for this particular data unit, E_{M_i}. This leads to:

$$Intlat_i(x,m) = \sum_{j=1}^{s}(r_{ij} * Slat_{nl_i(j)}) + Slat_{M_i} * (\sum_{j=1}^{s} w_{ij})$$

The delay incurred at a server E_j, $Slat_j$, consists of two components: (i) queueing time - the time spent by the request in the queue until it is served ($Qlat_j$) and (ii) the time spent in executing the query, which leads to:

$$Intlat_i(x,m) = \sum_{j=1}^{s}(r_{ij} * (Qlat_{nl_i(j)} + RLat)) + (\sum_{j=1}^{s} w_{ij}) * (Qlat_{M_i} + Wlat)$$

where $RLat$ and $WLat$ are the time taken by databases to execute read and update queries respectively. Figure 1 shows that depending on the load of the edge server, the queueing time is bound to increase. We calculate this latency using the well known Little's law, which states that the queue latency at server E_j ($Qlat_j$) is given as the product of the number of requests that arrive at E_j and the time taken to execute them [17]. We thus have:

$$Qlat_j = RLat * rps_j(x) + WLat * wps_j(x,m)$$

where $rps_j(x)$ and $wps_j(x,m)$ are the number of reads and writes per second performed by E_j respectively, under configurations x and m.

It can be seen that the problem of finding the optimal replica placement x and master selection m requires an exhaustive search of all combinations of placement configurations and is of exponential complexity $O(n^s)$. The decision problem of optimal placement is easily seen to be in NP. Furthermore, previous works have shown that this problem is NP-complete by reducing it from minimum set-cover [16]. We skip the formal proof due to space limitations.

4. Placement Heuristics

Since the problem of optimal placement and master selection is NP-complete, this calls for well-designed heuristics. In this paper, we propose three families of heuristics to perform replica placement and master selection.

```
1 forall $D_i$ do
2     forall $j$, $1 \leq j \leq s$ do
3         if $((r_{ij} + w_{ij}) \geq X/100 * (\sum_{j=1}^{s}(r_{ij} + w_{ij})))$ and $(r_{ij} > w_{ij})$ then
4             $x_{ij} \leftarrow 1$ ;
          else
5             $x_{ij} \leftarrow 0$ ;
          end
      end
6     If no replica can be selected according to rule (a), then
      select the server that offers the best $lat_i$ and place a
      replica there;
7     Select master ($M_i$) as the server that receives the largest
      number of updates among the selected replicas;
  end
```

Algorithm 1: Pseudocode of Fixed(X) algorithm

```
1 Run Fixed($X$) algorithms for different values of
  $X$=5,10,15,25,30;
2 Evaluate $lat$ of placement configurations obtained by each
  of these algorithms;
3 Choose the placement obtained by $Fixed(X)$ that yields the
  lowest value of $lat$;
```

Algorithm 2: Pseudocode of Adapt-Coarse algorithm

Fixed placement (Fixed(X)): In this heuristic, the system stores a replica of D_i at an edge server E_j if E_j's clients generate at least $X\%$ of the total requests to D_i. Once replicas are selected, it selects the replica that receives the

```
1 Sort data units in decreasing order based on the number of
  requests received by them ($\sum_{j=1}^{s}(r_{ij} + w_{ij})$);
2 repeat
3     Select the data unit $D_k$ that receives the most number of
      requests and has not yet been placed;
4     Evaluate $lat_k$ for placement configurations obtained by
      different $X$ values of Fixed(X), (e.g.,$X$=5,10,15,25,30);
5     Choose configuration $\{(x_{ij},m_{ij})\}_{i=k}$ that yields
      minimum $lat_k$;
6     Update the total number of reads and writes performed
      by each edge server holding a replica. (Note that this
      step affects the $lat$ calculation of subsequent data units);
  until all data units are placed;
```

Algorithm 3: Pseudocode of Adapt-Fine algorithm

most number of writes as the master for data unit. Obviously, the value of X determines the performance and different X values perform well for different system load and data access patterns. The pseudocode of this heuristic algorithm is given in Algorithm 1. Note that this heuristic is oblivious to the update characteristics of data units and the overall system load parameters while determining placement positions.

Adapt-Coarse (AC): The previous heuristic has the underlying limitation that the correct value of X needs to be determined manually. As noted earlier, this choice should be based on several factors such as a data unit's read-write ratio and access patterns. So, determining a good value manually may be difficult. This problem is alleviated by the AC heuristic. It automatically selects X by running the Fixed(X) algorithm and computing $lat(Fixed(X))$ for different values of X (X=5,10,15,25,30). Subsequently, it selects the value of X that obtains the lowest lat. The pseudocode of this heuristic algorithm is given in Algorithm 2. Note that this heuristic takes system load into account while calculating $lat(x, m)$. Once a value, X_{opt}, has been chosen, $Fixed(X_{opt})$ is applied to all data units.

Adapt-Fine (AF): The Fixed and AC heuristics fix the maximum replication degree of all data units at a coarse-grained level, irrespective of the characteristics of individual data units. With the AF heuristic, different data units can be treated differently based on their individual access patterns. The underlying idea behind the heuristic is to select the correct threshold value X for each data unit based on its own access characteristics and popularity. The algorithm works as follows: First, the system sorts data units so that it can place data units that receive the highest load first. For each data unit, D_i, the system evaluates the lat_i obtained for different X values of Fixed(X) given the load contributed by the already placed data units and selects the one that offers the least lat_i as the best placement for D_i. The system iterates this process for all data units. The pseudocode of this heuristic is given in Algorithm 3. The algorithm is greedy in nature, as it allocates more resources to popular data units expecting they can improve the overall system performance. We believe that this heuristic can produce good placements by taking into account the $Intlat$ in the calculation of lat_i and by deciding on placement at data-unit level.

Network latency (NW): This heuristic is similar to AF except that it optimizes only the $Nlat$ and assumes $Intlat$ to be negligible. In the computation of lat_i, $Intlat_i$ is set to 0. We use this heuristic as a baseline to compare the relative performance of other heuristics.

5. Evaluation of Heuristics

We evaluate the performance of different heuristics for different access patterns using simulations. We simulate a CDN with 100 edge servers handling 1000 data units. We study the effect of the following parameters on the performance of the proposed heuristics:

- **Number of data accesses (Load):** The average request rate received by each edge server. i.e., Load = $(\sum_{i=1}^{n}\sum_{j=1}^{s}(r_{ij} + w_{ij}))/(s * t)$, where t is the simulated duration of the experiment.
- **Write ratio (WR):** The ratio of the number of write accesses to total number of accesses performed on a data unit: $WR_i = \sum_{i=1}^{s} w_{ij} / \sum_{i=1}^{s}(r_{ij} + w_{ij})$
- **Data unit popularities ($alpha$):** We assigned popularities of data units according to Zipf distribution. This distribution takes one parameter $alpha$[1]. Earlier studies have shown that Web objects popularity follow a Zipf distribution [4, 11]. In our experiments, we vary $alpha$ from 0 (where all data units receive the same number of requests) to 3 (where very few data units receive most of the requests).

We assume that the distribution of requests to a data unit across edge servers follows Zip-f distribution and fix its parameter to 1. We assume the network latency between edge servers to be 500ms. We fixed the latency to execute a read query ($RLat$) and a write query ($WLat$) to 5 ms and 50ms respectively. These are the average latencies obtained from a PostgreSQL database server in Pentium-III 800 Mhz linux machine. Note that we performed our experiments for different values of network latencies. In all our experiments, the general trend observed in the relative performance of the evaluated heuristics were similar. Hence, we believe that the results presented in this section is not affected to a large extent by the constants used in our simulations.

In the following, we study the influence of each of the above parameters in isolated experiment settings. In each experiment, we vary only one parameter and fix the rest.

1 A Zipf distribution states that the frequency of occurrence of a particular value i is given by $f_i = C \cdot r_i^{-alpha}$ where r_i is the rank of i's occurrence.

Unless explicitly stated, we fix $alpha$ to 1, which is a typical value observed in real-world traces [11]. We fix WR to 30% and Load to 10 (which corresponds to an under-loaded system).

For each experiment, we plot the relative performance as the ratio of total latency obtained by a given heuristic to that of the NW heuristic. A lower value therefore denotes a better performance. We also plot the average replication degree to better illustrate the behavior of each heuristic. Each experiment was run 1000 times and the average results are reported.

5.1. Effect of $Load$

In our first experiment, we study the performance of different heuristics for varying values of Load. The results of our experiment are given in Figures 2(a) and (b). As seen in Figure 2(a), the AC and AF strategies perform better than NW and yield low latencies compared to NW for all load scenarios. This is because unlike NW, these strategies reduce the replication degree by considering the $Intlat$ in the computation of lat (see Figure 2(b)). Recall that more replication leads to higher load on individual servers (as 30% of accesses are updates).

Between the AF and AC heuristics, it can be seen that AF performs better as it is greedy in nature and begins placing replicas to more popular data units first, while selecting the replication degree on a data-unit level. The latter decides on the maximum replication degree and locations for all data units uniformly. Recall that since $alpha$ is fixed to 1, relatively few data units receive significant part of the load. AF being greedy in nature and by deciding placement at data-unit level performs better than AC. Furthermore, as seen in Figure 2(b), the AF strategy adapts its replication degree based on the system load.

5.2. Effect of WR

In this experiment, we study the performance of different heuristics for different values of the write ratio. The results of our experiment are given in Figures 2(c) and (d). As seen in the figures, the AF and AC heuristics perform significantly better than NW when there are large number of updates. This is because in these cases the $Intlat$ increases as the number of updates increases. So, by taking $Intlat$ into account during placement, the AF and AC strategies reduce the replication degree, thereby resulting in a better performance. With a read-only workload, all three heuristics perform similarly as $Intlat$ is very low in such cases. Again, the AF strategy chooses the correct point in the spectrum between partitioning and replication based on the WR, and yields the best performance.

5.3. Effect of $alpha$

In this experiment, we studied the performance for different values of $alpha$. The results are given in Figure 3.

Again for all values of $alpha$, AC and AF perform better than NW. AC performs better than AF for low values of $alpha$, where all data units receive roughly equal load. This is because the AF strategy, being greedy in nature, optimistically replicates for the first few data units assuming they constitute a significant portion of the workload. However, since all data units are equally popular this leads to poor replication for other data units thereby leading to overall poor performance. In this scenario, the AC heuristic performs better by fixing the replication degree globally for all data units.

In the case of higher values of $alpha$ where the popularity distribution of data units is skewed, a greedy heuristic such as AF performs better than AC (see Figure 3). This is because in such scenarios it is necessary to fix replica placement at a per-data-unit level rather than uniformly. Earlier research findings show that data popularities tend to exhibit high values of $alpha$, i.e., up to 1.4 [11]. Hence, we believe AF will yield better performance in real-world traces.

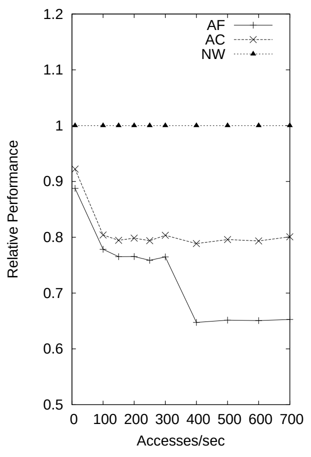
5.4. Discussion

The above experiments show that the AF heuristic performs better than its AC and NW counterparts for most of the workloads. This demonstrates that placement of data units must take into account the individual characteristics of data units and the overall system load. Of course, the problem of server load can be alleviated to some extent by adding more servers to the system and then partitioning data across the cluster of servers. Even though this is desirable in the long term, it is often not feasible to immediately add more servers this way. Moreover, when dealing with small organizations, it may not even be realistic to extend such a site. In such scenarios, the only alternative is to utilize the existing set of resources to the fullest extent and our placement heuristics allow one to make best use of existing resources.

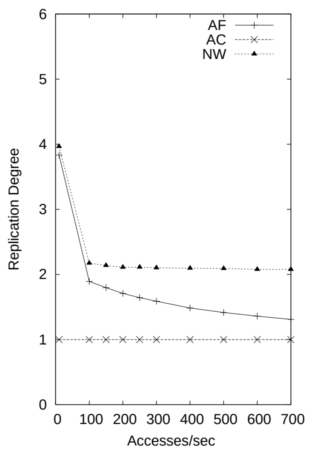
6. Related Work

The problem of replica placement has been widely studied in the context of static Web pages. Content placement problem deals with finding the best set of edge servers to host a replica such that an objective function (e.g., client latency and/or update traffic) is optimized. This problem is closely related to the problem studied in this paper and several works have addressed this in the context of static Web pages [16, 15]. However, as noted earlier, these algorithms assume that updates are rare assuming that replicating data reduces the load of each involved server.

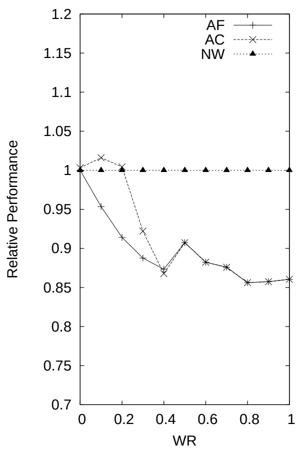
Commercial database caching systems such as DBCache [3] and MTCache [9] cache the results of selected queries and keep them consistent with the underlying database. These systems could be used to improve the performance of our system for applications that have a significant number of complex queries. Such approaches offer performance gains provided the data accesses con-



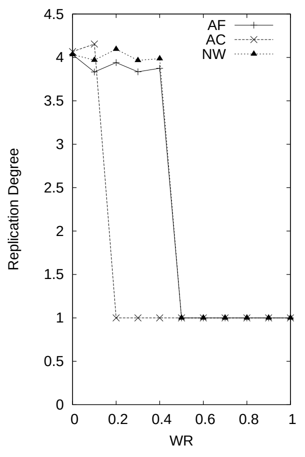
(a) Effect of $Load$ on Relative Performance



(b) Effect of $Load$ on replication degree

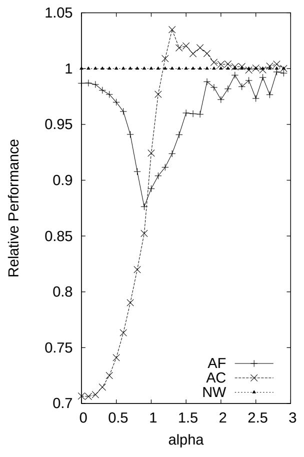


(c) Effect of WR on Relative Performance

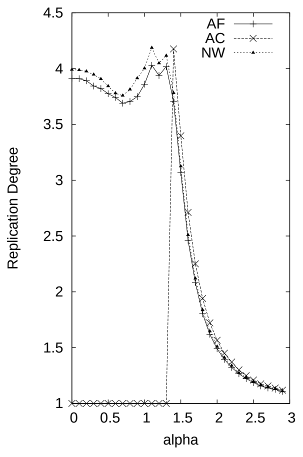


(d) Effect of WR on replication degree

Figure 2. Effect of $Load$ and WR



(a) Effect of $alpha$ on Relative Performance



(b) Effect of $alpha$ on replication degree

Figure 3. Effect of $alpha$

tain few write requests and have high temporal locality in read requests. However, the success of these schemes depends on the ability of the database administrator to identify the correct set of queries to cache. This requires careful manual analysis of the data access patterns.

7. Open Issues

In this section, we identify some of the crucial issues that need to addressed to build a truly autonomic CDN as we envisage here.

Routing Information. In order to answer a request to D_i, the data driver of each edge server should know where D_i's closest replica is located (or who its master is). Assuming that this information is relatively stable, spreading this information across servers should not cause a big communication overhead. However, maintaining this information can be a storage bottleneck for large database tables. We plan to address this by maintaining a *partial rout-*

ing table, which will store routing information for data units that are accessed often. In such settings, when a request is made to a data unit whose routing information is not locally available, the request can be forwarded to the origin server.

Update Traffic. In a CDN, it is desirable to optimize the system not only to reduce the request latency but also to reduce this traffic (and therefore the replication costs). In this paper, we explicitly ignored this issue because we just wanted to study the effect of different parameters (such as Load, RR) on client-perceived latency. However, as we demonstrated in [12], the tradeoff between update traffic and latency can be addressed in an automated fashion using a cost function that represents overall system performance into a single abstract figure.

Data granularity. Throughout this paper, we assumed that data unit replication is done at a fine-grain, such as at the level of database records. This choice allows us in theory to assign a different placement strategy to each data unit, which can lead to optimal performance [12]. However, such fine-grained replication can also result in significant overhead as the system must maintain replication information and select placement for each record individually. To address this, we propose to employ clustering techniques to group records with similar access pattern into coarse-grained data units so that placement can be performed on a lower number of data units with a little loss in performance.

Modelling Server Load. In this paper, we assumed a simple queueing model and used Little's law to model the queueing latency at each server. Of course, this model has limitations as it assumes an uniform stream of requests and need not model database load accurately. We plan to look at advanced models for this problem and use them in our placement heuristics.

Complex Queries. A limitation of our system is that it cannot handle complex queries as they require an exhaustive search of the database and these queries need to be sent to the origin server. To overcome this limitation, we plan to use template-based query approaches, such as [2], to cache the results of popular complex queries at edge servers thereby reducing the load of the origin server.

Failure Handling. In this paper, we assumed that the system is free of server and network failures. However, these failures may create a consistency problem if the master for a data unit is unreachable. We plan to address the issue of handling server failures (such failure of origin server, replica server and master server), in the near future.

8. Conclusions and Future Work

In this paper, we studied the problem of data unit placement for update-intensive Web applications in the context of GlobeDB. Our hypothesis is that there exists a continuous spectrum of placement choices between complete partitioning of sets of data units across edge servers and full replication of data units to all servers. We proposed and evaluated different families of heuristics for this problem of replica placement. As we have shown in our experiments, the AF heuristic perform better than its counterparts because it takes into account both the individual characteristics of data units and the overall system load. This results in it being able to choose the correct point in this spectrum of placement choices. We believe that this AF heuristic can significantly improve the performance of replicated update-intensive applications as well as less demanding applications such as e-commerce applications. As we stated in the previous section, there are several open issues that still need to be addressed here to realize the autonomic CDN we envisage and we plan to address them in our next steps.

References

[1] Akamai Inc. Edge Computing Infrastructure.

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

[3] C. Bornhvd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, June 2004.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the implications of Zipf's law for web caching. In *Proceedings of 3rd International WWW Caching Workshop*, 1998.

[5] J. Challenger, P. Dantzig, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, 4(4), Nov 2004.

[6] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In *Proceedings of the 2002 ACM SIGMOD International conference on Management of data*, pages 97–108. ACM Press, 2002.

[7] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[8] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 391–400. Morgan Kaufmann Publishers Inc., 2001.

[9] P. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL server. *Data Engineering*, 27(2):27–33, June 2004.

[10] 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 on World Wide Web*, pages 587–598. ACM Press, 2003.

[11] V. N. Padmanabhan and L. Qui. The content and access dynamics of a busy web site: findings and implicatins. In *SIGCOMM*, pages 111–123, 2000.

[12] G. Pierre, M. van Steen, and A. S. Tanenbaum. Dynamically selecting optimal distribution strategies for Web documents. *IEEE Transactions on Computers*, 51(6):637–651, June 2002.

[13] S. Sivasibramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *Proc. of the 14th International World-Wide Web Conference*, Chiba, Japan, May 2005.

[14] Speedera Inc. `http://www.speedera.com`.

[15] M. Szymaniak, G. Pierre, and M. van Steen. Latency-driven replica placement. In *Proceedings of the International Symposium on Applications and the Internet (SAINT)*, Trento, Italy, Feb. 2005.

[16] X. Tang and J. Xu. On replica placement for QoS-aware content distribution. In *Proc. IEEE INFOCOM'04*, March 2004.

[17] K. S. Trivedi. *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons Ltd., 2002.