# GlobeTP: Template-Based Database Replication for Scalable Web Applications

Tobias Groothuyse
Vrije Universiteit
De Boelelaan 1081a
Amsterdam, The Netherlands
tobias.groothuyse@xs4all.nl

Swaminathan Sivasubramanian
Vrije Universiteit
De Boelelaan 1081a
Amsterdam, The Netherlands
swami@cs.vu.nl

Guillaume Pierre
Vrije Universiteit
De Boelelaan 1081a
Amsterdam, The Netherlands
gpierre@cs.vu.nl

## ABSTRACT

Generic database replication algorithms do not scale linearly in throughput as all update, deletion and insertion (UDI) queries must be applied to every database replica. The throughput is therefore limited to the point where the number of UDI queries alone is sufficient to overload one server. In such scenarios, partial replication of a database can help, as UDI queries are executed only by a subset of all servers. In this paper we propose GlobeTP, a system that employs partial replication to improve database throughput. GlobeTP exploits the fact that a Web application's query workload is composed of a small set of read and write templates. Using knowledge of these templates and their respective execution costs, GlobeTP provides database table placements that produce significant improvements in database throughput. We demonstrate the efficiency of this technique using two different industry standard benchmarks. In our experiments, GlobeTP increases the throughput by 57% to 150% compared to full replication, while using identical hardware configuration. Furthermore, adding a single query cache improves the throughput by another 30% to 60%.

## Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of systems**]: Design studies; H.3.4 [**Information Storage and Retrieval**]: Systems and Software.

## General Terms

Performance.

## Keywords

Database replication, partial replication, scalability, Web applications.

## 1. INTRODUCTION

Over the past few years the World-Wide Web has taken significant importance into our lives, and many businesses and public services now rely on it as their primary communication medium. This drives the need for scalable hosting architectures capable of supporting arbitrary levels of load with acceptable performance. However, while this problem is now well understood for static content [11, 14, 20, 22], providing scalable infrastructures for hosting dynamically generated Web content still remains a challenge.

Dynamic Web content allows Web sites to personalize the delivered contents to individual clients, and to take action upon certain requests such as processing an order in an e-commerce site. Content is dynamically generated upon each client request by application-specific business logic, which typically issues one or more queries to an underlying database.

Numerous systems for scalable hosting of Web applications have been proposed. These systems typically cache (fragments of) the generated pages [7], distribute the computation across multiple application servers [1, 23] or cache the results of database queries [2, 4, 19, 27, 30]. However, although these techniques can be very effective depending on the application, in many cases their ultimate scalability bottleneck resides in the throughput of the origin database where the authoritative version of the application state is stored. Database replication techniques can of course help here, but the generic replication algorithms used by most databases do not scale linearly as they require to apply all update, deletion and insertion (UDI) queries to every database replica. The system throughput is therefore limited to the point where the quantity of UDI queries alone is sufficient to overload one server, regardless of the number of machines employed [12]. The only solutions to this problem are to increase the throughput of each individual server or to use partial replication so that UDI queries can be executed at only a subset of all servers. However, partially replicating a database is in turn difficult because queries can potentially span data items which are stored at different servers. Current partially replicated solutions rely on either active participation of the application programmer [15] or on one special server holding the full database to execute complex queries (and thereby becoming the new throughput bottleneck) [26].

This paper presents GlobeTP, a database replication system that exploits the fact that the database queries issued by typical Web applications belong to a relatively small number of query templates [2, 19, 27]. A query template is a parametrized SQL query whose parameter values are passed to the system at runtime. Prior knowledge of these templates allows one to select database table placements such that each query template can be treated locally by at


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

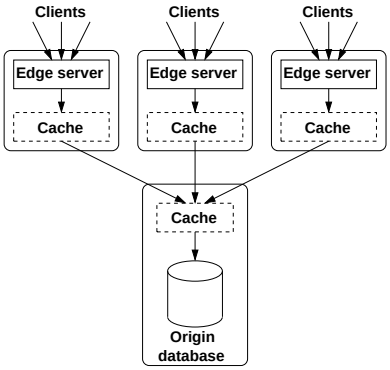



**Figure 1: Typical Edge-Server Architecture.**

least one server. We demonstrate that careful table placements based on the data span and the relative execution costs of different templates can provide major scalability gains in terms of sustained throughput. We further show that this technique can easily be used in combination with any existing template-based database query caching system, thereby obtaining reduced access latency and yet some more throughput scalability.

This paper is organized as follows: Section 2 presents the related work. Section 3 discusses our system model, and Section 4 details our table placement algorithms. Section 5 presents performance results and Section 6 discusses a number of issues that arise from our approach. Finally, Section 7 concludes the paper.

## 2. BACKGROUND AND RELATED WORK

Traditional content delivery networks (CDNs) often host Web applications using techniques such as fragment caching whereby (fragments of) the generated pages are cached at the edge servers [7, 10, 18]. This technique performs well if the temporal locality of requests is high and if the underlying database is updated rarely. However, applications that do not exhibit these behavior require more sophisticated edge-server-based techniques.

A typical edge-server architecture used by many advanced systems to host Web applications is depicted in Figure 1. Client requests are issued to edge servers located across the Internet. Each edge server has a full copy of the application code but no database. Database queries are sent out to a local query cache that can answer previously issued queries without incurring wide area latency. Cache misses and UDI queries are issued to the origin server. Queries are then potentially intercepted by another cache and, in the case of another miss, reach the origin database server to be processed. This architecture has been adapted in many versions depending on the specifics of each system.

The first type of edge-server architecture is edge computing infrastructures, where the application code is replicated at all edge servers and no cache is present [1, 23]. The database is kept centralized at the origin server so all database queries are forwarded to the origin server. Although this technique allows to distribute the computations to generate pages, it remains limited by the wide-area latency incurred for each query, and by the throughput bottleneck of the origin database.

To remove this bottleneck, various techniques have been proposed to cache the results of database queries at the edge servers [2, 4, 19, 27, 30]. Consistency of cached results must also be maintained when the underlying database is updated. This is usually done by requiring the application programmer to specify a number of query templates that are issued by the application. The programmer must also specify conflicts between templates. When a given UDI query template is invoked, all conflicting read templates are invalidated. This technique allows to reduce the database query latency as a number of queries can be answered locally. The total system throughput is also increased because less queries are addressed to the origin server. However, database caching systems have good hit ratio only if the database queries exhibit high temporal locality and contain relatively few updates.

A common technique used to improve the performance of a database is replication. Traditional RDBMS replication solutions replicate the complete database across several machines located within a cluster [6, 17, 21, 24]. The incoming read query workload is distributed evenly across all the replicas. Consistency is maintained by applying UDI queries to all replicas using 2-phase commit protocols or snapshot isolation. Database replication improves throughput as the incoming read query workload is shared among multiple servers. However, if the workload contains a significant fraction of UDI queries, then these systems incurs a limited throughput as all UDI queries must be applied to all replicas. As we show in our experiments, when the load of UDI queries alone is sufficient to overload any one of the servers, then the system cannot improve its throughput any more.

A number of commercial systems such as Oracle keep database servers consistent not by executing UDI queries at each replica but by executing these queries at the master server only, and by propagating update logs to the slave servers. Applying such logs is significantly more efficient than re-executing the queries. However, this technique does not improve the maximum throughput as it requires a single master server to execute all UDI queries. The master server then determines the total system's throughput.

A few database systems based on peer-to-peer technologies have been proposed to distribute the query processing load across arbitrary number of servers [9, 13]. These systems typically aim to achieve scalability by partioning the dataset across peers. However, the features offered by these systems are rather limited and handling complex queries that span the entire data set remains a challenge.

In [15], the authors propose an edge computing infrastructure where the application programmers can choose the data replication and distribution strategies that are best suited for the application. This approach can yield considerable gains in performance and availability, provided that the selected strategies are well suited for the application. However, coming up with the right strategies requires significant insight of the application programmers in domains such as fault-tolerance and weak data consistency. Contrary to this approach, we strive for requiring minimum support from the application programmers, and try to keep replication as transparent to the application as possible. Note that the main focus of this paper is not replication for availability. We however return briefly to this issue in Section 6.2 to

show how availability constraints can be taken into account in GlobeTP.

Some of our previous research already proposed to employ partial database replication to improve the performance of Web applications [26]. However, this architecture relies on record-level replication granularity. This design choice offers excellent query latency, but does not improve on throughput as a central server must maintain a copy of the full database (and therefore constitutes the throughput bottleneck of the system). Note that replication for throughput and replication for latency do not contradict each other. We show in Section 5.5 how GlobeTP can easily be coupled with a wide-area database caching system so that both throughput and latency can be improved at the same time.

# 3. SYSTEM MODEL

## 3.1 Application Model

Web applications are usually implemented by some business logic running in an application server, which is invoked each time an HTTP request is received. This business logic, in turn, may issue any number of SQL queries to the underlying database. Queries can be used to read information from the database, but also to update, delete or insert information. We refer to the latter as UDI queries.

We assume that the queries issued by a given Web application can be classified as belonging to a relatively small number of query templates. A database query template is a parametrized SQL query whose parameter values are passed to the system at runtime. This scheme is deployed, for example, using Java's prepared statement. Examples of parametrized read query templates include $QT1$: "SELECT price, stock, details FROM book WHERE id=?" and $QT2$ : "SELECT price, stock, details FROM book, author WHERE book.name LIKE (?) AND author.name = ?". An example of update template is $UT1$: "UPDATE price=price+1 FROM book WHERE id=?". In our current implementation, we require the application developer to explicitly define the templates. However, our implementation can be easily extended to apply static analysis techniques to identify the query templates automatically [19].

The explicit definition of query templates is at the basis of several database caching systems as it allows an easy definition of cache invalidation rules [2, 19, 27]. In contrast, the work presented in this paper uses the same notion of query templates but in a different manner: we exploit the list of templates to derive table placements that guarantee that at least one server is able to execute each query from the application.

## 3.2 System Model

The aim of our work is to increase the scalability of the origin database depicted in Figure 1 in terms of the maximum throughput it can sustain, while maintaining reasonable query execution latency. As shown in Figure 2, in our system the origin database is implemented by an array of database servers. We assume that all origin database servers are located within a single data center. The replication granularity in our system is the database table, so each database server hosts a replica of one or more table(s) from the application.

Since not all servers contain all the data, it is necessary to execute each query at a server that has all the necessary data

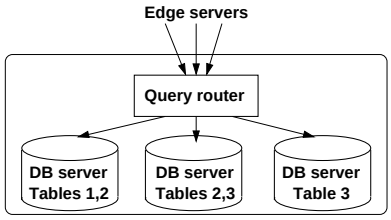



**Figure 2: Architecture of a partially replicated origin server.**

locally. This is ensured by the *query router*, which receives all incoming queries and routes each query to a server that contains all the necessary tables to answer the query. To this end, the query router knows the current placement of tables onto databases servers. It is also in charge of maintaining the consistency of the replicated databases. It issues UDI queries to all the servers that hold the tables to be modified; if all queries are successful then the operation is committed, otherwise it is rolled back. This simple algorithm guarantees sequential consistency.

In our implementations, all read and write queries are first received by the query router which in turn executes the query at appropriate replica. Since all UDI queries are queued in a single location at the query router, the router servers a serialization point and maintains consistency across replicas. Note that the current implementation of our system does not support transactions. Instead, GlobeTP treats each read and write query as independent operations. We believe that this is not a major restriction, as most Web applications do not require transactional database behavior. However, should transactions be required, adding support for them in our system would be relatively straightforward. Since the query router receives all incoming queries before they are executed, it can also act as the transaction monitor and implement any classical protocol such as two-phase commit. We however do not investigate this issue further in this paper.

Our work focuses on the structure of a Web application's origin database. Consequently, we make no assumption about the origin of the queries addressed to our system. In the simplest setup, queries can be issued directly by one or more application server(s). However, our system can also be easily coupled with a distributed database query cache such as DBProxy [2] and GlobeCBC [27]. In this case, the same definition of query templates can be used both by the caching system in order to maintain consistency, and the origin database in order to optimize throughput.

## 3.3 Issues

The work presented in this paper is motivated by the observation that the explicit definition of query templates of a Web application allows to select the placement of partially replicated data such that the total system throughput is optimized. We consider that such knowledge allow us to avoid two pitfalls that generic replicated databases necessarily face. First, application-unaware database systems do not know in advance the full set of query templates that will be issued to them. In particular, this means that it is impossible for them to determine a priori which tables must be kept together, and which ones can be sepa-

rated. Generic database systems usually address this issue by supporting only full replication, so that the data necessary to answer any query are always available at the same place. However, this has an important impact on the system's throughput. Second, the middleware that determines which replica should treat each read query does not take query characteristics into account. However, the execution times of different queries issued by a given Web application may vary by several orders of magnitude. In such a context, simple round-robin algorithms may not lead to optimal load balancing.

To be able to determine the placement of database tables on replica servers that allows to sustain the highest throughput, we must solve three main issues. First, not all possible placements of tables onto servers will allow to find at least one server capable of executing each of the application's query templates. We therefore need to analyze the set of query templates to determine a subset of placements that are functionally correct. Second, we must take the respective query execution times of different templates and their classification as read or UDI queries to determine the best placement in terms of throughput. Besides requiring accurate estimations of query execution times, finding the optimal placement incurs a huge computational complexity, even for relatively small systems. We therefore need a good heuristic. Finally, once the resulting system is instantiated we need to define a load balancing algorithm that allows the query router to distribute read queries efficiently across the servers that can treat them.

# 4. DATA PLACEMENT

The underlying idea behind our approach is to partially replicate the database so that UDI workload can be split across different replicas. This process involves the following three steps: (i) *Cluster Identification:* the process by which we determine the set of database tables that needs to be replicated together, (ii) *Load Analysis:* the process by which we determine the load received by each of the cluster, and (iii) *Cluster Placement:* determining the placement of the identified clusters across the set of database servers so that the load incurred by each of the database replica is minimized.

## 4.1 Cluster Identification

Our system relies on placement of individual tables on database servers to minimize the number of servers that must process UDI queries. However, not all placements are functionally correct as all tables accessed by a query template must be present in the same server for the query to be executed. The goal of cluster identification is to determine sets of tables that must be placed together on at least one server, such that there is at least one server where each query template can be executed.

We must characterize each query template with two attributes: (i) whether it is a read or a UDI query; (ii) the set of tables (also called table cluster) that it accesses. For instance, in the aforementioned query templates, template $QT1$ will be associated to a single-table cluster $\{book\}$, while $QT2$ will be associated to $\{book, author\}$. Clusters can overlap, as a table can belong to multiple clusters.

The problem of finding functionally correct table placements can then be reduced to a cluster placement problem; any table cluster placement will be functionally correct.

## 4.2 Load Analysis

Even though any cluster placement will lead to a functionally correct system, not all placements will lead to the same system throughput. To maximize throughput, it is crucial that no database server is overloaded. In other words, we need to place the table clusters such that we minimize the load of the most loaded server. This process is done in two steps. First, we evaluate the load imposed on each of the identified clusters for a representative workload. Second, we identify the placement that will create the best repartition of load across the servers.

### 4.2.1 Estimation of Load on Table Clusters

The load that each table cluster will impose on the server(s) where it is hosted depends on three factors: (i) the classification as belonging to a read or UDI template: read queries can be executed on one server, while UDI queries must be applied on all servers holding the corresponding cluster; (ii) the occurrence frequency of the template in the expected workload; and (iii) the computational complexity of executing the query on a given database server. Classifying queries as read or UDI can be done by simple query analysis. Similarly, the occurrence frequency of templates can be derived from observation of an existing workload. However, estimating the load that each query imposes on the database where it is run requires careful attention.

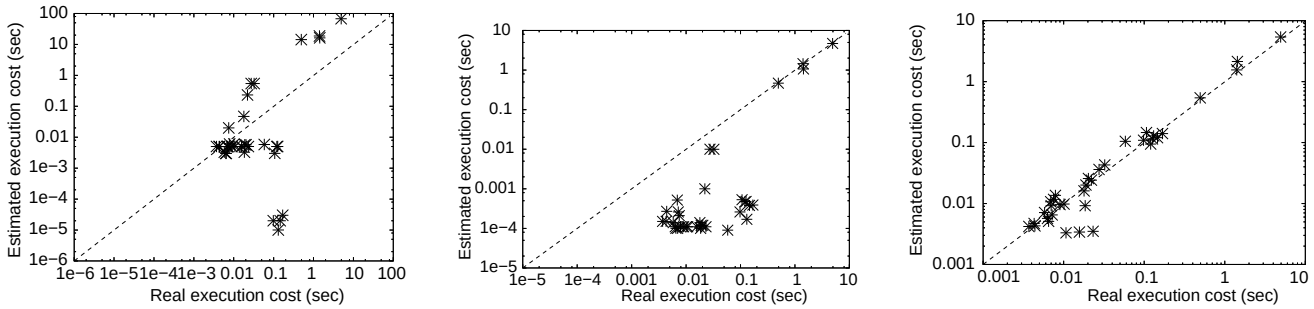
Mature database systems such as MySQL and PostgreSQL make their own estimations of the internal execution of different queries as part of their query optimization procedure. These execution time estimations are made available, for example using PostgreSQL's EXPLAIN <query> and EXPLAIN ANALYZE <query> commands. However, these estimations take only the actual execution time into account, and ignore other factors such as the connection overhead. Another possible method consists of simply measuring the response time of each query template in a live system. The advantage of this method is that it measures the end-to-end response time of the database tier and includes connection overhead. Note that query execution times should be measured under low load to avoid polluting measurements with load-related overheads such as the queuing latency [29].

Figure 3 shows the accuracy of the three cost estimation techniques applied to the query templates from the RUBBoS benchmark [25]. In each graph we estimated the cost of each query template and compared it with the actual execution time under high load. A perfect estimator would produce points located on the $y = x$ diagonal line. Clearly, the estimations produced by the database query optimizers are not as accurate as actual measurements made under low loads. In the rest of this paper we therefore restrict ourselves to this last method.

### 4.2.2 Estimation of Load on Database Servers

In a replicated database, read queries are executed at one database node, whereas UDI queries are executed at all nodes that hold the data modified by the UDI query. To determine the load that each database server will incur for a given table placement and a given query workload, we must distinguish the two types of queries.

Each UDI query in the studied workload will result in applying the associated execution cost to each of the database servers holding the corresponding table(s). On the other hand, each read query will create execution cost on only




(a) EXPLAIN query. (b) EXPLAIN ANALYZE query. (c) Measurement under low load.

**Figure 3: Accuracy of different methods for query cost estimation.**

```
1 Distribute table clusters uniformly across server nodes;
2 S = set of server nodes ;
3 while S ≠ ∅ do
    /*We want to minimize the maximum server load*/
4   while (max(estimated server load) is decreased) do
5     N = Most loaded server in S ;
6     foreach Table cluster C placed in N do
7       Try to reduce N's load by migrating or
        replicating C to other servers;
      end
    end
8   S = S - (the most loaded server in S);
  end
```

**Algorithm 1**: Pseudocode of the table cluster placement algorithm.

one server; we count that, on average, each database server holding the corresponding cluster will incur the execution cost of the query, divided by the number of replicas.

This analysis allows us to roughly compute the execution cost that each database server will incur for a given table placement and a given query workload. To maximize the system throughput, it is essential that no database server is overloaded. We therefore aim at balancing the load such that the cost of the most loaded server is minimized.

## 4.3 Cluster Placement

Finding the optimal table placement can be realized by iterating through all valid table placements, evaluating the respective cost of each database server under each placement, and selecting the best one. However, the computational complexity of this exhaustive search is $O(2^{N*T}/N!)$, where $T$ is the number of table clusters to be placed and $N$ is the number of nodes to place them on. This very high complexity makes it unpractical even for relatively small system sizes. We must therefore find a heuristic instead.

As shown in Algorithm 1, our heuristic starts with a very roughly balanced placement, and iteratively tries to improve it by applying simple transformations in table placement.

The first step (step 1) is to place clusters uniformly onto servers to create an initial configuration. The heuristic then iteratively attempts to improve the quality of the placement (steps 3-8). Since the goal is to find the placement where the maximum server load is minimized, we identify the most loaded server (step 5) and try to offload some of its clusters to other servers (steps 6 and 7). Two techniques can be used here (step 7): either migrating one of the clusters to another server (thereby offloading the server of the whole associated load), or replicating one of the clusters to another server (thereby offloading the server of part of the read query load). The algorithm evaluates all possible operations of this type, and checks if one of them improves the quality of the placement. This operation is repeated until no more improvement can be gained (step 4). The most loaded server, which cannot be offloaded any more without overloading another one is considered to be in its 'optimal' state and is removed from the working set of servers (step 8). The algorithm then tries to optimize the load of the second most loaded server, and so on until all servers have reached their 'optimal' state.

Even though there is no guarantee that this heuristic will find the optimal placement, in our experience it always identifies a reasonably good placement within seconds (whereas the full search algorithm would take days).

## 4.4 Query Routing

Query routing is an important issue that affects the performance of replicated databases. Simple round-robin schemes are efficient only when all the incoming queries have similar cost. However, when applications tend to have queries with different execution costs, round-robin scheduling can lead to load skews across database servers, resulting in poor access latencies.

Query routing gains a higher significance in GlobeTP. In a partially replicated database system such as ours, queries can no longer be sent to arbitrary database nodes. The query router used in GlobeTP thus differs from the traditional query routers used in fully replicated databases in the following aspects. First, read queries can be scheduled only among a subset of database servers instead of all servers. Second, UDI queries must executed at all database servers that store the tables modified by the incoming UDI query.

The process of selecting a database server to route an incoming read query has considerable impact on the overall performance of the system. This is usually determined by the routing policy adopted by the replication system. In our work, we experimented with the following policies.

### 4.4.1 RR-QID: Round-Robin per Query ID

RR-QID is an extension to the round-robin policy that is suitable for partially replicated databases. In this policy, the query router maintains a separate queue for each

query template identified by its query identifier, QID. Each queue is associated with the set of database servers that can serve the incoming queries of type QID. Subsequently, each incoming read query is scheduled among the candidate servers (associated with its queue) in a round-robin fashion.

#### 4.4.2 Cost-Based Routing

The underlying idea behind the cost-based routing policy is to utilize the execution cost estimations to balance the load among database servers. To this end, upon arrival of an incoming query, the query router first estimates the current load of each database server. Subsequently, it schedules the incoming query to the least loaded database server (that also has the required set of tables).

To this end, the query router maintains a list of queries that have been dispatched to each database server and still awaiting response. This list contains the list of queries currently under (or awaiting) execution at a database server. Subsequently, the load of a database server is approximated as the sum of the estimated cost of these queries. Finally, the server with least cost is scheduled to execute the next incoming query.

We show the respective performance of these two routing policies in the next section.

## 5. PERFORMANCE EVALUATION

In this section, we compare the performance of full database replication to GlobeTP for two well-known Web application benchmarks. TPC-W is a standard e-commerce benchmark that models an online bookstore such as `amazon.com` [28], while RUBBoS is a bulletin-board benchmark modeled after `slashdot.org` [25]. We selected these two applications for their different data access characteristics. For example, in a typical news forum, most users are interested in the latest news. On the other hand, in a bookstore application, the shopping interests of customers can be very diverse, thereby leading to much lower query locality. This allows us to study the behavior of our systems for different data access patterns. In addition to these experiments, we also evaluate the benefit of adding a database query caching layer to GlobeTP.

### 5.1 Experimental Setup

The TPC-W application uses a database with seven tables, which are queried by 23 read and 7 UDI templates. In our experiments the database is initially filled with 288,000 customer records. Other tables are scaled according to the TPC-W requirements. TPC-W defines three standard workload mixes that exercise different parts of the system: 'browsing' generates 5% update interactions; 'shopping' generates 20% update interactions; and 'ordering' generates 50% update interactions. We use the 'shopping' mix, which results in a database workload containing 5.6% of UDI queries[1].

The RUBBoS application consists of a set of PHP scripts and a database containing five tables regarding users, stories, comments, submissions and moderator activities. The database is initially filled with 500,000 users, out of which 10% have moderator privileges, and 200,000 comments. The size of the database is approximately 1.5 GB. The application defines 36 read and 8 UDI query templates. In our experiments, we used the default user workload which generates 0.76% of UDI queries.

The client workload for both applications is generated by Emulated Browsers (EBs). The run-time behavior of an EB models a single active client session. Starting from the home page of the site, each EB uses a Customer Behavior Graph Model (a Markov chain with Web pages acting as nodes and navigation action probabilities as edges) to navigate among Web pages and perform a sequence of Web interactions. The behavior model also incorporates a think time parameter that controls the amount of time an EB waits between receiving a response and issuing the next request, thereby modeling a human user more accurately. We set the average think time to 6 seconds.

To generate flexible yet reproducible workloads, we run each benchmark under relatively low load (i.e., with 30 to 100 EBs) multiple times and collect the corresponding database query logs. Query logs from different runs can then be merged to generate higher load scenarios. For instance, to evaluate the performance of our system for 600 EBs, we merge the query logs from six different runs with 100 EBs, and stream the result to the query router. This allows us to study the performance of the database tier alone independently of other tiers.

The query router is implemented as a stand-alone server written in Java. It maintains a pool of connections to each database server and schedules each incoming query based on the adopted routing policy. Database servers run PostgreSQL version 8.1.3. Both full and partial database replication are performed at the query router level as described in Section 3.2.

All our experiments are performed on a Linux-based server cluster. Each server is configured with dual-processor Pentium III 900 MHz CPU, 2 GB of memory and a 120 GB IDE hard disk. These servers are connected to each other with a gigabit LAN, so the network latency between the servers is negligible.

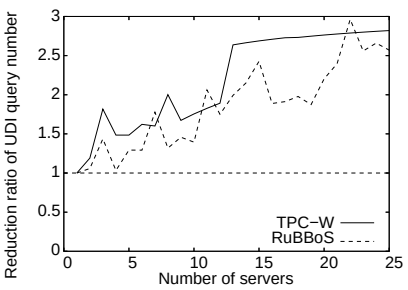
### 5.2 Potential Reductions of UDI Queries

One important goal of GlobeTP is to reduce the replication degree of individual database tables to reduce the number of UDI queries to be processed. However, the extent to which this is feasible greatly depends on the query templates issued by the application and the workload distribution.

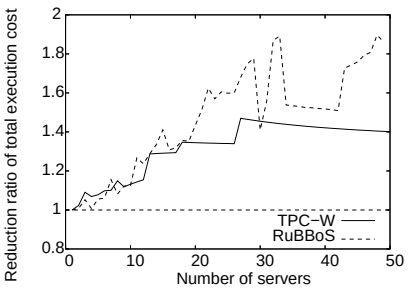
Figure 4(a) shows to which extent table-level partial replication allows to reduce the number of UDI queries to be processed, expressed as the ratio of UDI queries to execute between full and partial replication. The higher the ratio, the greater the gain. Obviously, with just one server to host the database, partial and full replication are identical so the ratio is equal to 1. As the number of servers increases, partial replication allows to reduce the number of UDI queries by a ratio close to 3 for both applications.

To evaluate more accurately the potential load reduction that we can expect from partial replication, we should take into account the respective estimated costs of different query templates, as well as the read queries from the workload. Figure 4(b) shows the reduction ratio of estimated costs imposed on each server, for different numbers of servers. As we can see, the reduction factor is much lower than when count-

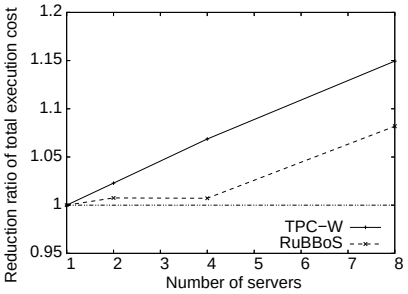
[1]Note that one must distinguish the update interactions from the UDI queries. Update interactions are user-generated HTTP requests that lead to at least one UDI query, plus any number of read queries. Since GlobeTP only operates at the database query level, the proportion of update interactions is irrelevant here.



(a) Reduction of the number of UDI queries.



(b) Reduction of estimated execution cost per server.



(c) Reduction of estimated execution cost per server (zoom).

**Figure 4: Potential Reduction of UDI queries.**

ing UDI queries alone. The reason is that both workloads are dominated by read queries, which are equally spread in full and partial replication.

The experiments described in the remaining of this paper are based on configurations using up to 8 database servers. Figure 4(c) shows the respective potential of partial replication for both benchmarks under these conditions. TPC-W shows a relatively good potential, up to 15% reduction in workload per server. On the other hand, RUBBoS has a lower potential. This is mainly due to the fact that RUBBoS generates very few UDI queries; reducing their number even further by ways of partial replication can therefore have only a limited impact.

Note that other workloads may show somewhat different behavior. For example, RUBBoS defines a workload where search queries are disabled. Since searches are implemented as very expensive read queries, removing them from the workload mechanically improves the cost ratio of UDI queries and thereby the gains to be expected from GlobeTP. Conversely, we cannot exclude that other Web applications may dictate to keep all database tables together, making our form of partial replication equivalent to full replication. For these, GlobeTP will not provide any improvement unless the application itself is updated (see Section 6.1).

In this paper we focus on standard benchmarks which offer real yet limited potential for use in our system. However, as we will see in the following sections, even the relatively modest reductions in estimated costs shown here allow for significant gains in execution latency and in total system throughput.

## 5.3 Effect of Partial Replication and Template-Aware Query Routing

To illustrate the benefits of GlobeTP, we measured the query execution latencies of read and UDI queries together using different configurations. For each of the two benchmarks, we compared the performance of full replication, GlobeTP using RR-QID query routing, and GlobeTP using cost-based query routing. In all cases we used 4 database servers and one query router. We selected a load of 900 EBs for TPC-W and 330 EBs for RUBBoS, so that the tested configurations would be significantly loaded.

Figure 5 shows the cumulative latency distributions from both sets of experiments. As one can see, in both cases GlobeTP processes queries with a much lower latency than full replication. For example, in RUBBOS GlobeTP processes 40% more queries than full replication within 10 ms. In TPC-W, GlobeTP processes 20% more queries within 10 ms than full replication.

In TPC-W, the RR-QID query routing policy delivers better performance than its cost-based counterpart. This can be explained by the fact that in TPC-W the costs of different query templates are relatively similar. The unavoidable inaccuracy of our cost estimations therefore generates unbalanced load across servers, which leads to sub-optimal performance. On the other hand, RR-QID is very effective at balancing the load when queries have similar cost.

In RUBBoS, GlobeTP combined with cost-based routing outperforms both other configurations. In this case, the costs of different queries vary by three orders of magnitude (as shown in Figure 3(c)). In this case, cost-based routing works well because even relatively coarse-grained estimations of the cost of each query helps avoiding issuing simple queries to already overloaded servers.

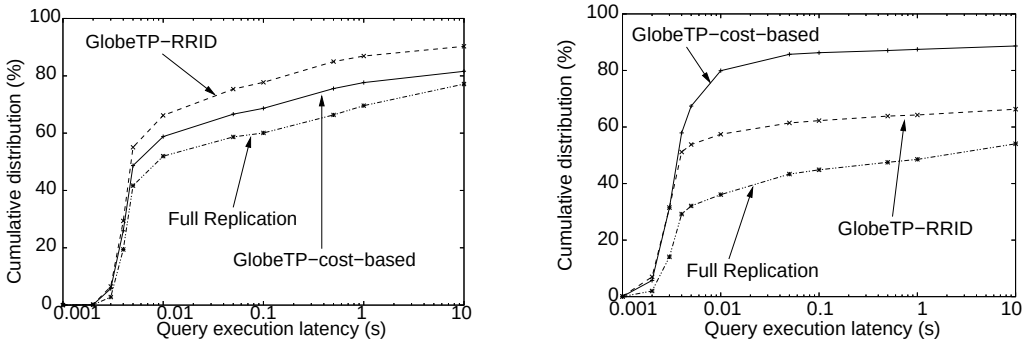
In the following experiments we restrict ourselves to the most effective routing policy for each application. We therefore use RR-QID for measurements of TPC-W, and cost-based routing for RUBBoS.

One should note that GlobeTP has greater effect on the latency in the case of RUBBoS than for TPC-W. This may seem contradictory with results from the previous section. However, the latency and the throughput of a given system are not necessarily correlated. As we will see in the next section, the throughput improvements that GlobeTP provides are significantly greater for TPC-W than RUBBoS.

## 5.4 Achievable Throughput

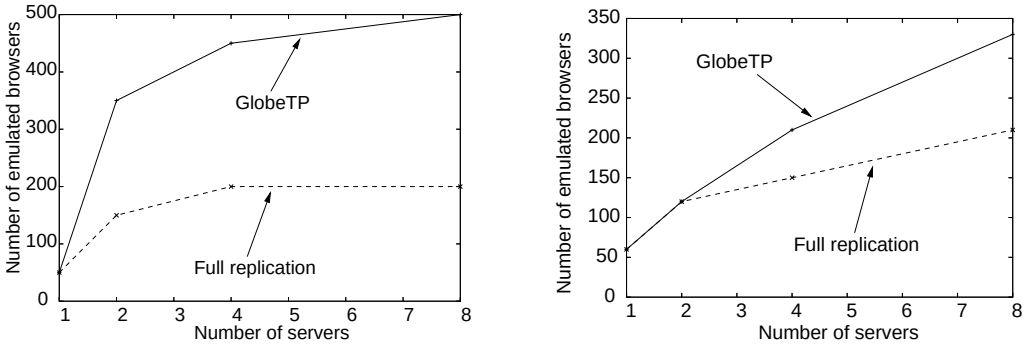
To evaluate the scalability of our approach, we measured the maximum sustainable throughput of different approaches when using identical hardware resources. We first set a performance target in terms of query execution latency: in our experiments we aim at processing at least 90% of database queries within 100 ms. Note that this performance target is quite challenging, as several query templates have execution times greater than 100 ms, even under low loads (see Figure 3(c)).

We then exercise system configurations with full and partial replication, and increase the workload by steps of 50 EBs for TPC-W and 30 EBs for RUBBoS. For each configuration we record the maximum number of EBs that each configuration can serve while respecting the latency target. The results are shown in Figure 6.




(a) TPC-W, 900 EBs. (b) RUBBoS, 330 EBs.

**Figure 5: Query latency distributions using 4 servers.**




(a) TPC-W. (b) RUBBoS.

**Figure 6: Maximum achievable throughputs with 90% of queries processed within 100ms.**

In TPC-W, one server alone can sustain up to 50 EBs. As we increase the number of database servers, partial replication performs significantly better than full replication. In particular, the maximum throughput of the fully replicated system does not improve with more than four servers. This corresponds to the point when the treatment of UDI queries alone saturates each server. This can be explained by the fact that the execution time of a UDI query is typically an order of magnitude higher than that of a simple read query. On the other hand, GlobeTP can sustain up to 150% higher throughput while using identical hardware resources. Unlike full replication, it is capable of exploiting 8 servers to sustain higher throughput than when using only 4. This is due to the fact that each server has less UDI queries to process, and thereby experiences lower load and better execution latency.

In RUBBoS, GlobeTP again performs better than full replication, yet with a lower difference. With 4 and 8 servers, GlobeTP sustains 120 more EBs than full replication, which accounts for 57% of throughput improvement. Given that RUBBoS generates very few UDI queries, little improvement can be gained by further reducing their number with partial replication. In this case, the major reason for throughput improvement is the cost-aware query routing policy which takes the relative costs of different queries into account to better balance the load between servers.

## 5.5 Effect of Query Caching

As noted in Section 3.2, GlobeTP can easily be coupled with a database query caching system as most query caching systems rely on the same assumption as GlobeTP regarding the explicit definition of query templates. However, GlobeTP focuses on improving the throughput of the application's origin database, while query caching systems aim at reducing the individual query execution latencies. We therefore consider that both types of system complement each other very well. As a side effect, a query caching system can also improve the system throughput, as it prevents a number of read queries from being issued to the origin database.

**Table 1: Maximum throughput of different configurations.**

| | TPC-W | RUBBoS |
|---|---|---|
| Full replication (4 servers) | 200 EBs | 150 EBs |
| GlobeTP (4 servers) | 450 EBs | 210 EBs |
| GlobeTP (4 servers) + 1 cache | 600 EBs | 330 EBs |

In our experiments, we use our own in-house query caching solution, GlobeCBC [27]. GlobeCBC acts as a very simple object cache: unlike other similar systems it does not attempt to merge multiple query results into a single view of the database. Instead, GlobeCBC stores the result of each query independently from the others. This allows for very fast processing, and facilitates the execution of the replacement policy when the size of the cached items exceeds a given limit. In our experiments, we limited the cache size to approximately 5% of the size of the database itself.

Table 1 shows the effect of adding a single cache server in front of the query router when using four database servers. In TPC-W, the cache had a hit rate of 18%. This relatively modest hit rate is due to the fact that the standard TPC-W workload has very low query locality compared to real e-commerce sites [3]. However, even in this case the system throughput is increased by 33%, from 450 to 600 EBs.

Unlike TPC-W, the RUBBoS workload has quite high database query locality. In this case the query cache delivers 48% hit ratio, which effectively increases the throughput by 57%, from 210 to 330 EBs. This result is quite remarkable considering that search queries, which are by far the most expensive ones in the workload, are based on random strings and are therefore always passed to the origin database for execution.

# 6. DISCUSSION

## 6.1 Potential of Query Rewriting

This paper demonstrates that relatively simple techniques allow to significantly improve the throughput of standard benchmarks, without requiring any modification to the applications themselves. However, we believe that increased throughput can also be gained from simple changes of the application implementation.

The main limitation of the approach of table-granularity partial replication comes from database queries that span multiple tables. As the business logic of an application becomes more complex, we can expect that more join queries are introduced. Such queries oblige the table placement algorithm to place all relevant tables together on at least one server, which in turn increases the replication degree and reduces the maximum throughput. Of course, queries spanning multiple tables are occasionally indispensable to the application. But we have observed from the TPC-W and RUBBoS benchmarks that many such queries can easily be rewritten as a sequence of simpler queries spanning one table each.

One simple example is the following query from TPC-W, which aims at obtaining the most recent order issued by a particular customer: "`SELECT o_id FROM customer, orders WHERE customer.c_id = orders.o_c_id AND c_uname = ? ORDER BY o_date, orders.o_id DESC LIMIT 1`." This query spans two tables. However the `customer` table is used here only to convert a full-text username into a user ID, after which the most recent order issued from this ID can be researched in the `order` table. It is then trivial to rewrite the application to first issue a query to `customer` table alone, then another one to search for the most recent order.

We found such unnecessarily complex queries to be very frequent in the applications that we studied. Rewriting them into multiple simpler queries can only reduce the constraints put on the table placements, and therefore result in higher throughput.

## 6.2 Fault-Tolerance

Although the main focus of this paper is not replication for availability, one cannot ignore this issue. With increased number of server machines involved in a given application, the probability that one of them fails grows quickly. However, the most general form of fault-tolerance for this kind of system cannot be realized, as providing both consistency and availability in the presence of network partitions is impossible [5, 16]. On the other hand, if we ignore the possibility of network partitions and restrict ourselves to server failures, then the problem has an elegant solution.

To guarantee that the partially replicated database remains able to serve all the expected query templates, it is essential that each query template be available at one or more servers. Therefore, to tolerate the failure of at most $N$ servers one only has to make sure that each query template is placed on at least $N + 1$ servers. This requires database replication algorithms suitable for fault-tolerance, which is a well-understood problem. Replicating the query router is also relatively simple as long as the applications do not need transactional guarantees. As the query router does not maintain any dynamically updated state that is essential for application correctness, no state consistency between multiple instances of the query router needs to be implemented.

Starting from a configuration designed for throughput only, planning for fault-tolerance can be done in two different ways. First, one may keep the number of servers unchanged but artificially increase the replication degree of table clusters across the existing machines. However, this will likely degrade system throughput as more UDI queries must be processed per server. Alternatively, one may provision for additional servers, and adjust table placement to keep the worst-case throughput constant. As long as not too many servers fail, this configuration will exceed its throughput requirements, which may have the desirable side-effect of protecting the Web site to a certain extent against unexpected variations in load.

# 7. CONCLUSION

In this paper we have presented GlobeTP, which exploits table-granularity partial database replication to optimize the system throughput. This solution relies on the fact that the query workload of Web applications is composed of a restricted number of query templates, which allows us to determine efficient data placements. In addition, the identification of query templates allows for efficient query routing algorithms that take the respective query execution costs to better balance the query load. In our experiments, these two techniques allow to increase the system throughput by 57% to 150% compared to full replication, while using identical hardware configuration.

To increase the system's scalability even further, a natural extension is to combine GlobeTP with a database query caching system such as GlobeCBC, as both systems rely on the same definition of query templates. These systems complement each other very well: query caching improves the execution latency in a wide-area setting by filtering out many read queries, while GlobeTP shows its best potential for improving throughput under workloads that contain many UDI queries. In our experiments, the addition of a single query cache allows to improve the achievable throughput by approximately 30% to 60%.

The work presented in this paper does not take into account the long-term load variations that must be expected when operating a popular dynamic Web site. Adapting the system capacity without any service interruption requires dynamic database provisioning, which is a very difficult problem that is only beginning to be addressed [8]. In the near future we plan to study whether knowledge of query templates can help here.

## 8. REFERENCES


[1] Akamai EdgeSuite. http://www.akamai.com/en/html/services/edgesuite.html.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proc. Intl. Conf. on Data Engineering*, pages 821–831, March 2003.

[3] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69, August 2001.

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

[5] E. A. Brewer. Towards robust distributed systems (abstract). *Proc. ACM Symp. on Principles of Distributed Computing*, July 2000.

[6] E. Cecchet. C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26, June 2004.

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

[8] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of databases in dynamic content web servers. In *Proc. Intl. Conf. on Autonomic Computing*, Dublin, Ireland, June 2006.

[9] Z. Chen, Z. Huang, B. Ling, and J. Li. P2P-Join: A keyword based join operation in relational database enabled peer-to-peer systems. In *Proc. Intl. Conf. on Database and Expert Systems Applications*, Sept. 2006.

[10] 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 *Proc. ACM SIGMOD/PODS Conf.*, pages 97–108, June 2002.

[11] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5), September-October 2002.

[12] B. Fitzpatrick. Inside LiveJournal's backend, or "holy hell that's a lot of hits!". Presentation at the O'Reilly Open Source Convention, July 2004. http://www.danga.com/words/2004_oscon/oscon2004.pdf.

[13] W. Fontijn and P. Boncz. AmbientDB: P2P data management middleware for ambient intelligence. In *Proc. PERWARE Workshop*, Mar. 2004.

[14] M. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. Symp. on Networked Systems Design and Implementation*, pages 239–252, San Francisco, CA, USA, March 2004.

[15] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proc. Intl. WWW Conf.*, May 2003.

[16] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[17] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 134–143, Cairo, Egypt, September 2000.

[18] 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 *Proc. Intl. WWW Conf.*, pages 587–598, May 2003.

[19] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. Maggs, and T. Mowry. A scalability service for dynamic web applications. In *Proc. Conf. on Innovative Data Systems Research*, pages 56–69, Asilomar, CA, USA, January 2005.

[20] G. Pierre and M. van Steen. Globule: a collaborative content delivery network. *IEEE Communications Magazine*, 44(8):127–133, August 2006.

[21] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proc. ACM/IFIP/USENIX Intl. Middleware Conf.*, Toronto, Canada, October 2004.

[22] M. Rabinovich and A. Aggarwal. RaDaR: a scalable architecture for a global web hosting service. In *Proc. Intl. WWW Conf.*, May 1999.

[23] M. Rabinovich, Z. Xiao, and A. Agarwal. Computing on the edge: A platform for replicating internet applications. In *Proc. Intl. Workshop on Web Content Caching and Distribution*, pages 57–77, Hawthorne, NY, USA, September 2003.

[24] M. Ronstrom and L. Thalmann. MySQL cluster architecture overview. MySQL Technical White Paper, April 2004.

[25] Rubbos: Bulletin board benchmark. http://jmob.objectweb.org/rubbos.html.

[26] S. Sivasubramanian, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *Proc. Intl. WWW Conf.*, Chiba, Japan, May 2005.

[27] 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, Amsterdam, The Netherlands, June 2006. http://www.globule.org/publi/GCBRCDWA_ircs022.html.

[28] W. D. Smith. TPC-W: Benchmarking an ecommerce solution. White paper, Transaction Processing Performance Council.

[29] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. ACM SIGMETRICS*, pages 291–302, June 2005.

[30] W. Zhao and H. Schulzrinne. Enabling on-demand query result caching in DotSlash for handling web hotspots effectively. In *Proc. Workshop on Hot Topics in Web Systems and Technologies*, Boston, MA, USA, November 2006.