

Analysis of Caching and Replication Strategies for Web Applications

Swaminathan Sivasubramanian¹ Guillaume Pierre¹
Maarten van Steen¹ Gustavo Alonso² *

Abstract

Replication and caching mechanisms are often employed to enhance the performance of Web applications. In this article, we present a qualitative and quantitative analysis of state-of-the-art replication and caching techniques used to host Web applications. Our analysis shows that the selection of best mechanism is heavily dependant on the data workload and requires careful analysis of the application characteristics. To this end, we propose a technique that will enable Web practitioners to compare the performance of different caching/replication mechanisms.

There are many reasons why Web sites can be slow and an important one is dynamic generation of Web documents. Modern Web sites such as Amazon.com and Slashdot.org do not simply deliver static pages but generate content on the fly each time a request is received, so that the pages can be customized for each user. Clearly, generating a Web page in response to every request takes more time than simply fetching static HTML pages from disk. The main cause is that generating a dynamic Web page typically requires to issue one or more queries to a database. Access times to the database can easily get out of hand when the request load is high.

A number of techniques have been developed in industry and academia to overcome this problem. The most straightforward one is Web page caching where (fragments of) the HTML pages generated by the application are cached for serving future requests [7]. For example, Content Delivery Networks (CDNs) like Akamai¹ deploy edge servers around the Internet, which locally cache Web pages and deliver them to the clients. By delivering pages from edge servers that are usually located close to the client, CDNs reduce the network latency for each request. Page caching techniques used in CDNs work well if many requests to the Web site can be answered with the same cached HTML page. These techniques have shown to be effective in hosting many Web sites [3, 7]. However, with growing drive towards personalization of Web sites, a generated page tends to be unique for every user, thereby reducing the benefits of conventional page caching techniques.

¹Vrije Universiteit, Amsterdam¹ and ETH Zurich, Switzerland². Email: {swami,gpierre,steen}@cs.vu.nl, alonso@inf.ethz.ch

¹<http://www.akamai.com>

The limitation of page caching techniques have triggered the CDN and database research community into investigating new approaches for scalable hosting of Web applications. These approaches can be broadly classified into four techniques: replicate application code [10], cache database records [1, 4], cache query results [8] and replicate the entire database [9, 6]. While numerous research efforts have been spent in developing each of these approaches, very few works have analyzed their pros and cons and examined their performance. The objective of this article is to present an overview of various such scalability techniques and present a comparative analysis of its features and performance. To do so, let us first consider well-known scaling techniques for Web applications.

Techniques to scale Web applications

Instead of storing dynamic pages after they are generated by some central Web server, various techniques aim at replicating the *means* to generate the pages over multiple edge servers. Despite their differences, these techniques often rely on the assumption that the applications do not require strict transactional semantics for their data accesses (as for example banking applications do). They typically provide “read-your-writes” consistency which guarantees that when an application at an edge server performs an update, any subsequent reads from the same edge server will return the effects of that update (and possibly others). Scalable techniques that do provide transactional semantics are beyond the scope of this article.

Edge computing

The simplest way to generate user-specific pages is to replicate the application code at multiple edge servers, and keep the data centralized (see Figure 1(a)). This technique is, for example, the heart of the Edge Computing product from Akamai and ACDN [10]. Edge computing (EC) allows each edge server to generate user-specific pages according to context, sessions, and information stored in the database, thereby spreading the computational load across multiple servers. However, the centralization of the data can also pose a number of problems. First, if edge servers are located worldwide, then each data access incurs wide-area network latency; second, the central database quickly becomes a performance bottleneck as it needs to serve all database requests from the whole system. These properties restrict the use of EC to Web applications that require relatively few database accesses to generate the content.

Data Replication

The solution to the database bottleneck problem of EC is obviously to place the data at each edge server so that generating a page requires only local computation and data access. Database replication (REPL) techniques can help here, by maintaining identical copies of the database at multiple locations [9, 6, 2]. However, in Web environments, the database replicas are typically located across a wide-area network, whereas most database replication middleware assumes the presence of a local-area network between

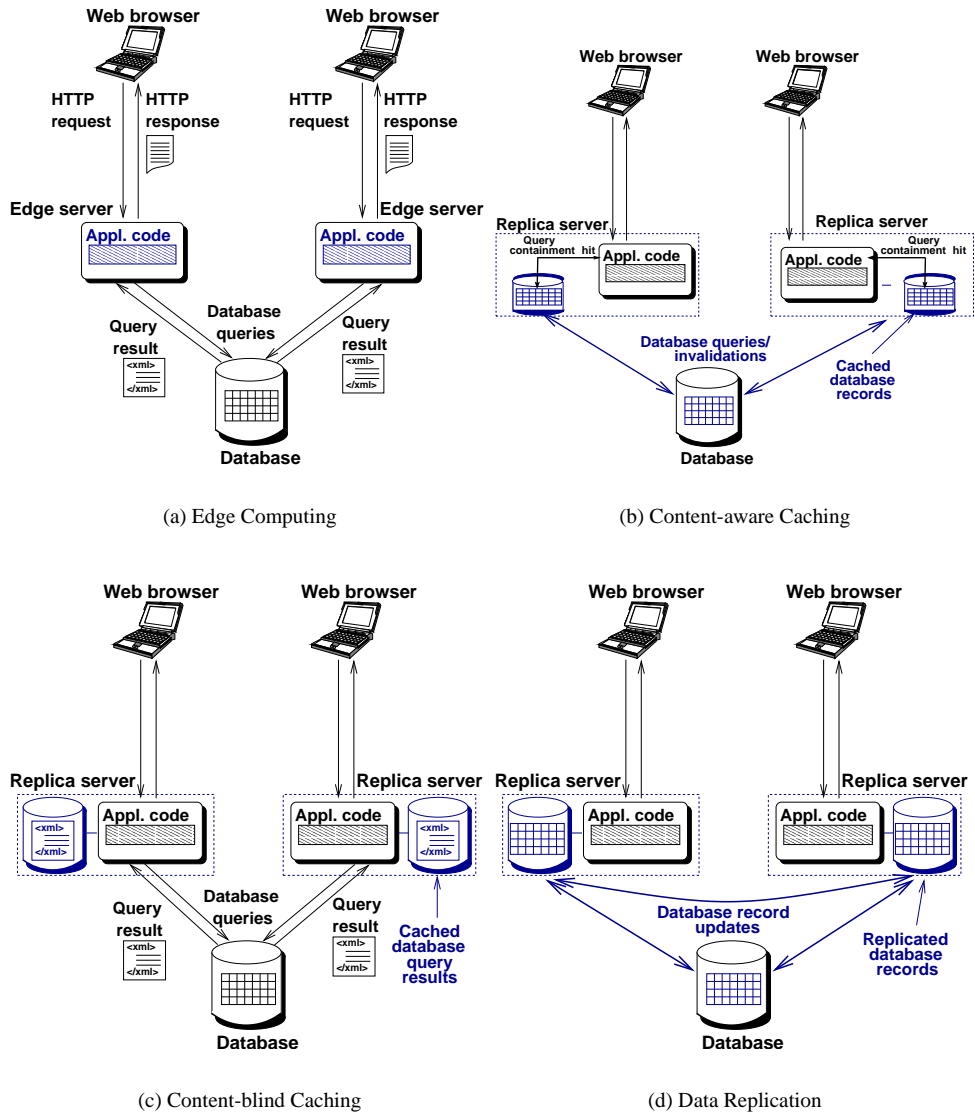


Figure 1: Variety of solutions that address problem of scalable Web hosting

replicas. This can be a problem if many database updates are generated by a Web application. In this case, each update needs to be propagated to all the other replicas to maintain the consistency of the replicated data, potentially introducing a huge network traffic and performance overhead. In our study, we designed a simple replication middleware solution where all updates are serialized at an origin server and propagated to the edges in a lazy fashion. Each read query is answered locally at the edges.

Content-aware data caching

Instead of maintaining full copies of the database at each edge server, content-aware caching systems (CAC) simply cache the result of database queries as they are issued by the application code. In this case, each edge server maintains a partial copy of the database. Each time a query is issued, the edge-server database needs to check if it contains enough data locally to answer the query correctly. This process is called query containment check. If the containment check result is positive, then the query can be executed locally. Otherwise, it must be sent to the central database. In the latter case, the result is inserted in the edge-server database so that future identical requests can be served locally. The process of inserting cached tuples into the edge database is done by creating insert/update queries on the fly and requires a good understanding of the application's data schema. Examples of CAC systems include DBCache [4] and DBProxy [1].

CAC allows to store query results in a storage-efficient way. For example, the queries “select * from items where price < 50” (Q1) and “select * from items where price < 20” (Q2) have overlapping results. By inserting both results into the same database, the overlapping records are stored only once. Another interesting feature of content-aware caching is that once the result of Q1 has been stored, Q2 can be executed locally, even though that particular query has never been issued before. In this case, the query containment procedure would recognize that the results of Q2 are contained in the results from Q1. CAC systems are beneficial when application's query workload has range queries or queries with multiple predicates (e.g., find items which satisfy $\langle clause1 \rangle$ OR $\langle clause2 \rangle$).

Typically, a query containment check is highly computationally expensive, as we need to check the new query with *all* previously cached queries. To reduce this cost, CAC systems utilize the fact that Web applications often consist of a fixed set of read and write query templates. A query template is a parameterized SQL query whose parameter values are passed to the system at runtime. Use of query templates can vastly reduce the search space of query containment as each incoming query needs to be checked only with a relatively small set of query templates. Using a template-based check, in the above example Q1 and Q2 might be checked only with other cached instances of the template QT1: “select * from items where price<?” and not with instances of, for example, QT2: “select * from items where subject=?”. However, this method can also reduce the cache hit rate. These systems use template-based mechanisms to ensure cache consistency. In CAC systems, the update queries are always executed at the origin server (the central database server). When an edge server caches a query, it subscribes to receive invalidations of conflicting query templates.

For instance, in the above example an update to change the price of an item table will conflict with QT1.

Content-blind data caching

An alternative to CAC is content-blind query caching (CBC). In this case, the edge servers do not need to run a database at all. Instead, they store the results of remote database queries independently [8, 12]. CBC uses a method akin to template-based invalidation to maintain the consistency of its cached results [12]. In CBC, since the query results are not merged together, caching the answers to the queries Q1 and Q2 defined above would lead to storing redundant information. Also, the cache will have a hit only if exactly the same query is issued multiple times at the same edge server. This can potentially lead to suboptimal usage of storage resources and lower cache hit rates. On the other hand, it has some advantages. First, in CBC, the process of checking if a query result is cached or not is trivial and incurs very little computational load. In contrast, query containment procedures in CAC is relatively expensive as it requires examination of at least a subset of cached queries. Second, by caching query results as result sets (e.g., by means of JDBC or PHP drivers) instead of database records, CBC system can return results immediately in case of a cache hit. In contrast, CAC pays the price of database query execution and can increase the load on edge servers. Third, inserting a new element into the cache does not require a query rewrite and requires merely storing objects.

Cache replacement policy is an important issue in any caching system as it determines which query results to cache and which ones to evict from the cache. An ideal cache replacement policy must take into account several metrics such as temporal locality, cost of the query, update characteristics of the database. Note that cache replacement in CBC is simple as each result is stored independently and many popular replacement algorithms can be applied here [12]. However, since CAC merges multiple query results, its replacement policy should ensure that removal of a query result does not remove the affect the results of other cached queries.

Performance Analysis

To make a quantitative comparison of these four techniques, we evaluate their performance for two different applications: RUBBoS, a bulletin board benchmark application that models `slashdot.org`, and TPC-W, an industry standard e-commerce benchmark that models an online bookstore like `www.amazon.com`.

RUBBoS application's database consists of five tables, storing information regarding users, stories, comments, submissions and moderator activities. We filled the database with information of 500,000 users and 200,000 comments. The TPC-W benchmarks consists of seven database tables. Its database is filled with information on 100,000 items and 288,000 customers. For our experiments, we choose the open source PHP implementation of these benchmarks². These two applications have very

²<http://jmob.objectweb.org/rubbos.html>, <http://pgfoundry.org/projects/tpc-w-php/>

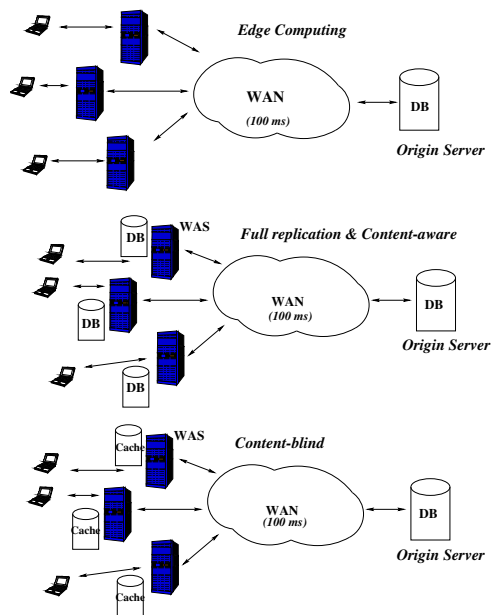


Figure 2: Architectures of evaluated systems

different data access characteristics. In a typical bulletin board, users are usually interested in the latest news and so the workload can exhibit high locality. On the other hand, in a bookstore application the shopping interests of customers may vary, thereby leading to lower query locality. This allows us to study the behavior of different systems for different data access patterns. However, it must be noted that these benchmarks are by no means a complete representative of e-commerce workload.

The client workload for both benchmarks is generated by Emulated Browsers (EBs) and conforms to the TPC-W specification for clients [13]. The average think time (i.e., the amount of time an EB waits between receiving a response and issuing the next request) is set to 5 seconds. The user workload for RUBBoS contains more than 15% interactions that lead to updates. For TPC-W, we study the performance for two kinds of workloads: browsing (95% browsing and 5% shopping interactions) and ordering workload (equal fraction of browsing and shopping interactions). For TPC-W, we modified the client workload behavior such that the book popularity follows a Zipf distribution (with $\alpha = 1$), which was found in a study that observed data characteristics of Amazon online bookstore [5].

Experiment Setup

For our tests, we used two servers with dual-processor Pentium III 900 Mhz CPU and 1 GB memory. We use the Apache 2.0.49 Web server with PHP 4.3.6 in our edge server, PostgreSQL 7.3.4 for our DBMS and PgPool for pooling database connections³. We

³<http://pgfoundry.org/projects/pgpool/>

emulated a wide-area network between an edge server and the origin server, by directing all network traffic to an intermediate router that runs the NISTNet network emulator⁴. This router delays packets sent between the different servers to simulate a realistic wide-area network. In the remaining discussion, we refer to links via NISTNet with a bandwidth of 50 Mbps and a latency of 100ms as WAN links, and 100 Mbps and zero latency as LAN links. Note that these bandwidth and latency values are considerably optimistic, as the Internet bandwidth usually varies a lot and is constantly affected by network congestion. These values are chosen to model the best network conditions for a CDN built on an Internet backbone and are the *least favorable* conditions to show the best performance of any data caching or replication system. The experimental setup of the evaluated systems are described in Figure 2. As can be seen, communication between the clients and the edge server happens through a LAN link. Even though this is unrealistic, we believe this is acceptable as the “client-to-edge” latency is the same for the evaluated systems.

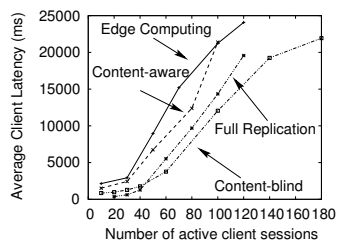
All experiments are started with a cold cache. The system is warmed up for 20 minutes, after which measurements are taken for a period of 90 minutes. As we did not want the effect of cache replacement algorithms to affect the performance of CAC and CBC, we did not constrain the storage capacity of the edge server, i.e., the size of the cache repository was not restricted. At the outset, this might look advantageous for caching systems, however in our experiments we observed that amount of disk space required for cache repository was at the most only 20% of the size of the entire database. In all experiments, we vary the request load (expressed in terms of number of active client sessions) and measure the end-to-end *client latency*, which is the sum of the *network latency* (the time spent by the request traversing the WAN) and *internal latency* (the time spent by the request in generating the query responses and composing the subsequent HTML pages).

Performance Results

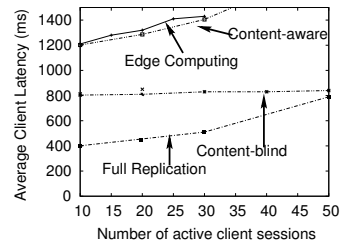
The results of our experiment are shown in Figure 3. For RUBBoS, CBC performs the best in terms of client latency (except under low loads) while edge computing performs the worst. The reason for CBC’s superior performance with RUBBoS is twofold. First, RUBBoS’s workload exhibits high temporal locality (yielding a cache hit ratio of up to almost 80%) thereby avoiding WAN latency. Second, the query execution latency incurred in generating a query response, for CBC, is much lower than that of REPL (or CAC) as the caching system avoids database query planning and execution latency. This allows CBC to sustain higher load than REPL and CAC. EC performs worse than the other architectures as each data access incurs a WAN latency and all requests are served by a single origin server. However, during low loads, REPL performs marginally better than CBC as each query is answered locally thereby avoiding any wide-area network latency. Moreover, during low loads, the internal latency incurred in generating a query response is lower than the network latency incurred in answering a query.

In our experiments, CBC and CAC have almost the same hit ratio, despite the

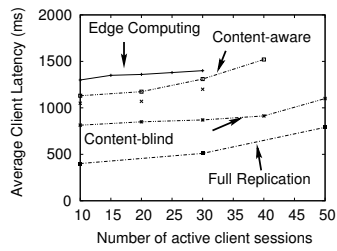
⁴<http://snad.ncsl.nist.gov/itg/nistnet/>



(a) RUBBoS benchmark



(b) TPC-W Browsing



(c) TPC-W Ordering

Figure 3: Performance of different architectures for RUBBoS and TPC-W application

fact that CAC's merged storage allows more hits than CBC. The reason is that CAC's merged storage is most beneficial for workloads with many range queries and queries with multiple predicates. However, in RUBBoS, most of the queries are exact lookup queries, which do not benefit from the flexibility offered by query containment tests. The increase in latency for CAC can also be attributed to the increased overhead of query containment, cache management (inserting and invalidating caches), query planning, and execution.

For TPC-W, EC performs the worst in client latency while REPL performs the best (see Figures 3(b) and (c)). In this case, CBC and CAC perform relatively poor because the TPC-W benchmark workload exhibits poor temporal locality which yields a hit ratio of at most 35% in our experiments. REPL performs better as each query can be answered locally. CBC performs better than CAC as the former's cache hit ratio is only marginally less compared to the latter. Again, this is again due the fact that TPC-W does not fully exploit CAC's query containment features. Due to space constraints we do not present detailed discussions of our results. For extensive discussions and more results on the performance of these systems for multiple edge servers and using weak consistency mechanisms, we refer interested readers to [12].

Discussion

From the experiments, we can see that there is no clear winner. For applications whose query workload exhibits high locality (e.g., RUBBoS), CBC performs the best. CAC does not perform as well as expected mostly because the tested query workload do not fully exploit the query containment features of CAC. For applications that have a predominant load of such queries, we believe CAC will outperform CBC systems. For applications that exhibit poor locality (such as TPC-W benchmark), data replication schemes perform better than content-blind caching. Our conclusion is that there exists no single solution that can perform the best for all Web applications and workloads.

Choosing the right strategy for your web application

Since different techniques are optimal for different applications, Web designers should choose them by carefully analyzing the characteristics of the Web application. The question is *how to choose the correct strategy?*. In general, the best strategy is the one that reduces the end-to-end client latency of the application. The end-to-end client latency is affected by various parameters such as hit ratio of caches (page cache, CAC or CBC), application server execution time, and database query execution time⁵. To estimate the end-to-end latency of a Web application, we need to estimate these parameters. While parameters such as execution time of application servers and databases can be measured by server instrumentation and log analysis, measuring the cache hit ratio for different caching strategies such as CBC and CAC is harder. This is because it is not always feasible to deploy multiple caching systems just for measuring their cache hit ratio. Ideally, we would like to measure the possible hit ratio of different

⁵For a detailed description of the model to estimate end-to-end latency of a multi-tiered Web application, we refer interested readers to our earlier work [11].

caching strategies without having to run each of them. To this end, virtual caches may be deployed.

Virtual Cache

A virtual cache (VC) behaves like a real cache except that it stores only the metadata such as the list of objects in the cache, their sizes, invalidation parameters. Objects themselves are not stored. By applying the same operations as a real cache, VCs can estimate the hit rate that would be offered by a real cache with same configuration. Since a VC stores only the metadata, it requires less memory. For example, to measure the hit ratio of a cache that can hold millions of data items, the size of a virtual cache required will be in the order of only a few megabytes. We can for example use a pair of such virtual caches to determine the effective hit ratio of CAC and CBC. Note that for implementing a virtual CAC, one has to implement the query containment checker into the VC in addition to simple put, get, and invalidation operations. By running such VCs, a Web administrator can determine the hit ratios (HRs) of different caching techniques.

By definition, the hit ratio of a VC should be same as a real cache as both perform the same operations. Our experiments with virtual GlobeCBC, fragment caches and CAC also confirm this. Moreover, compared to static trace-driven analysis, VC is more effective due to its ability to perform instant and on-line measurement of cache hit ratios. By running VCs for different caching techniques and different cache sizes, the administrators can readily compare the benefits of different caching techniques (such as CAC, CBC and page caches). Using these estimated hit ratios of different caching techniques and other parameters such as execution times of application server and database server, one can estimate the end-to-end response time of the application using the model we described in [11]. Subsequently, the best performing caching/replication technique can be chosen as the one that provides the least end-to-end latency.

Note that this process of selection can be done during the initial phases of application deployment. Subsequently, this decision can be revised periodically, if necessary. In such cases, different VCs must be run (with the application) only during the decision making period. However, if the application experiences frequent change in workload then one can envisage running these VCs continuously and performing adaptations more frequently, as described in [11].

Conclusions

Various caching and replication techniques are available for improving the performance of Web applications. Our analysis shows that there is no single best replication/caching solution for Web applications. Application designers/administrators need to monitor (continuously) the query workload of their applications to decide on the solution that is best suited for their application. To this end, we propose an online technique based on virtual caches that can be used to compare the relative performance of different caching techniques. We believe this technique (in combination with the analytical model de-

scribed in [11]) can aid administrators in choosing the best caching/replication technique for their applications.

References

- [1] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In *Proceedings of International Conference on Data Engineering*, pages 821–831, 2003.
- [2] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, 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, 2001.
- [4] C. Bornhvd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, June 2004.
- [5] E. Brynjolfsson, Y. J. Hu, and M. D. Smith. Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. *Manage. Sci.*, 49(11):1580–1596, 2003.
- [6] E. Cecchet. C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26, June 2004.
- [7] J. Challenger, P. Dantzic, and K. Witting. A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, 4(4), Nov 2004.
- [8] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *Proceedings of CIDR*, pages 56–69, 2005.
- [9] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, Toronto, Canada, Oct. 2004.
- [10] M. Rabinovich, Z. Xiao, and A. Agarwal. Computing on the edge: A platform for replicating internet applications. In *Proceedings of the Eighth International Workshop on Web Content Caching and Distribution*, pages 57–77, Hawthorne, NY, USA, Sept. 2003.
- [11] S. Sivasubramanian, G. Pierre, and M. van Steen. Towards autonomic hosting of multi-tier internet applications. In *Proceedings of the USENIX/IEEE HotAC-I Workshop*, June 2006.

- [12] 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, 2006.
- [13] W. Smith. TPC-W: Benchmarking an e-commerce solution. http://www.tpc.org/tpcw/tpcw_ex.asp.