# Holistic Query Transformations for Dynamic Web Applications

Amit Manjhi*, Charles Garrod†,
Bruce M. Maggs†‡, Todd C. Mowry†, Anthony Tomasic†
*Google, Inc. †Carnegie Mellon University ‡Akamai Technologies

## Abstract

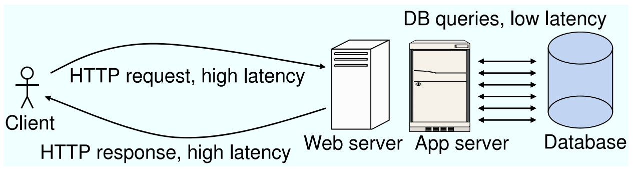
*A promising approach to scaling Web applications is to distribute the server infrastructure on which they run. This approach, unfortunately, can introduce latency between the application and database servers, which in turn increases the network latency of Web interactions for the clients (end users). In this paper we introduce the concept of source-to-source holistic transformations—transformations that seek to optimize both the application code and the database requests made by it, to reduce client latency. As examples of our concept, we propose and evaluate two source-to-source holistic transformations that focus on hiding the latencies of database queries. We argue that opportunities for applying these transformations will continue to exist in Web applications. We then present algorithms for automating these transformations in a source-to-source compiler. Finally, we evaluate the effect of these two transformations on three realistic Web benchmark applications, both in the traditional centralized setting and a distributed setting.*

## 1. Introduction

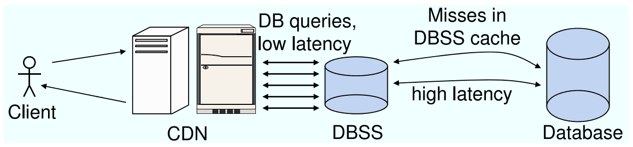
Anyone on the Internet can access a Web application. As a result, Web applications suffer from unpredictable load, particularly due to breaking news (e.g., Hurricane Katrina) or popularity spikes (e.g., the Slashdot effect). To address the scalability challenge, Web applications increasingly use a distributed infrastructure. The distributed infrastructure inevitably introduces latency between the different tiers of the application, which in turn increases the latency experienced by users. User studies [1, 2] have shown that high user latencies drive customers away. Therefore user latencies must be kept low even when using a distributed architecture.

To ensure low user latencies, it is important to understand how this latency arises. A Web application is a collection of programs. On an HTTP request, an application server runs one or more of these programs to generate the response. These programs, in turn, issue database queries to obtain the data needed for generating the response. Frequently, the programs issue multiple database queries for each HTTP interaction: e.g., for the benchmark applications we study, the average number of queries per dynamic HTTP response varies between 1.8 and 9.1.

In a traditional centralized setting, these database queries are answered by a database server that is in the same administrative domain and connected to the application server(s) by a high bandwidth, low latency link. As a result, these multiple



(a) Latency in a traditional centralized architecture.



(b) Latency in a distributed architecture.

**Figure 1: Latency in traditional vs. distributed architectures.**

round-trips have little impact on the overall latency a user experiences. The user latency is dominated by the high latency of reaching the web server of the application. Figure 1(a) shows the different latency components in a traditional centralized setting.

In a distributed setting, an application may use geographically distributed Content Delivery Network (CDN) nodes to scale its web and application servers and geographically distributed Database Scalability Service (DBSS) nodes to scale its database [5, 7]. The database queries issued by an application server are handled by a DBSS node, which attempts to answer these queries from its query-result cache. If the request hits in the DBSS cache, the delay in obtaining the query result is minimal. However, if the request misses in the cache, the user must endure the delay in getting the response back from the home server database. This delay is typically long because the scalability service nodes are geographically distributed. Figure 1(b) shows the different latency components in a scalability service setting. Even after methods to boost the cache hit rate are employed by scalability service nodes, users are likely to experience high latency if multiple database requests miss the cache on an HTTP request.

To reduce the client latency, it is desirable to either eliminate database requests or hide their latencies. There are several reasons why opportunities to do so appear in current Web applications. First, these applications are typically written for a traditional centralized setting, in which there is minimal overhead in issuing multiple database requests. Not expecting a distributed environment, application developers frequently do not optimize for the number of database requests the application issues. Second, application developers often abstract database values as objects in the program, a paradigm that is also adopted by Object Relational Mapping tools [3, 9]. If they need multiple values, they just issue multiple queries. Third, there are instances where it is easier for developers to express their main logic in the procedural language because it is closer to how the data is actually presented to the user.

In this work we propose two transformations that rewrite the application code to either eliminate database requests or hide their latencies. Our first transformation, the MERGING transformation, eliminates queries by clustering related queries. Our second transformation, the NONBLOCKING transformation, hides the long latency in fetching query results, by overlapping the execution of queries.

Web applications are commonly written in a procedural language like Java or PHP whereas they issue database queries in a declarative language, typically SQL. Both transformations that we propose change the database queries as well as the application code surrounding them. They affect the program as a whole. Therefore we call them *holistic* transformations. To evaluate their effectiveness, we have applied it to three benchmark applications. While we currently applied them manually, we believe that the algorithms (described in the extended technical report version of this paper [6]) should be straightforward to automate in a source-to-source compiler [4, 8]. We also defer the detailed discussion of these two transformations and the related work to the technical report [6].

## 2. The MERGING Transformation: Clustering Related Queries

We explain the MERGING transformation using an illustrative example. Consider the code fragment in Figure 2(a), which is taken from the AUCTION benchmark. The program issues several short related queries and then combines their results. In a DBSS setting, for each query that results in a cache miss at the DBSS node, the user must endure the long delay of accessing the home server database. Assuming a constant hit rate at the DBSS cache, the client latency is proportional to the number of queries issued in an HTTP interaction. The MERGING transformation transforms the code to the equivalent code in Figure 2(b), merging all of the short inter-related queries into one join query. The program then needs to issue just one query instead of the previous $N+1$ queries, assuming the loop is repeated $N$ times.

In the technical report [6], we discuss the effect we expect the MERGING transformation to have on the total work done by the system, identify the code patterns to which we can apply the transformation, and describe our algorithms for implementing the transformation.

```
$template:=SELECT from_user_id
          FROM comments
          WHERE to_user_id = ?;
$query:=set_params ($template, $to_id);
$result:=execute ($query);
foreach ($row in $result) {
  $from_id:=get_user_id ($row);
  $template:=SELECT user_name
            FROM users
            WHERE user_id = ?;
  $query:=set_params ($template, $from_id);
  $result2:=execute ($query);
}
```

(a) Original code

```
$template:=SELECT from_user_id, user_name
          FROM comments, users
          WHERE from_user_id = user_id
           AND to_user_id = ?;
$query:=set_params ($template, $to_id);
$result:=execute ($query)
```

(b) After the MERGING transformation

**Figure 2: A code fragment from the AUCTION application, before and after applying the MERGING transformation. The code, an example of the Loop-to-join pattern, finds the names of users who have posted comments about a particular user. We focus on two base relations: `users` with attributes `user_id` and `user_name`, and `comments` with attributes `from_user_id` and `to_user_id`.**

## 3. The NONBLOCKING Transformation: Prefetching Query Results

After issuing a database query, a Web application waits for the query result. In some cases this wait is unnecessary because the next database query does not depend on the answer to the current query. In such cases, the client latency can be greatly reduced by overlapping the query executions. In this section we present the NONBLOCKING transformation, which can overlap executions of multiple queries that do not depend on each other by "prefetching" query results.

To illustrate how this transformation can be applied to a code fragment, consider Figure 3, which shows two functionally equivalent code fragments from the BOOKSTORE application. Figure 3b shows the code after applying the NONBLOCKING transformation. The method $execute\_non\_blocking$ does not block and only serves to populate the cache with the query result. If the latency of the first database request is $t_a$ and the latency of the second request is $t_b$, this transformation reduces the overall latency from $t_a + t_b$ to $\max\{t_a, t_b\}$.

Ideally, whenever the program that dynamically generates the HTTP response starts running, we would like to issue prefetch requests for all queries that the program will issue during its execution. However, issuing a prefetch request for each query, at the start of the program's execution, is not always possible because: (1) one of the parameters of the query may be the result of a previous query, (2) the query may be conditionally issued and the condition uses the result of a pre-

```
$template1:=SELECT item_name
          FROM items i1, items i2
          WHERE i1.id = i2.related
           AND i2.id = ?;
$query1:=set_params ($template1, $id);
$result1:=execute ($query1);
$template2:=SELECT user_name FROM users
          WHERE user_id = ?;
$query2:=set_params ($template1, $user_id);
$result2 := execute ($query2);
```

(a) Original code

```
$template2:=SELECT user_name FROM users
          WHERE user_id = ?;
$query2:=set_params ($template2, $user_id);
execute_non_blocking ($query2);
$template1:=SELECT item_name
          FROM items i1, items i2
          WHERE i1.id = i2.related
           AND i2.id = ?;
$query1:=set_params ($template1, $id);
$result1:=execute ($query1);
$result2:=execute ($query2);
```

(b) After the NONBLOCKING transformation

**Figure 3: A simplified code fragment from the BOOKSTORE application, which finds the name of an item related to the item the user is viewing and the name of the user, given her id. We focus on two base relations: `users` with attributes `user_id` and `user_name`, and `items` with attributes `item_id`, `item_name`, and `related`.**

vious query, and (3) there may be an update statement before the query that may affect the query result.

Formally, each program of a Web application can be represented as a directed acyclic graph, where the nodes are database accesses, and there is an edge between two nodes if one node has to be executed after the other node for correctness. Given this directed acyclic graph, a database access can be issued as soon as all database accesses that are its ancestors in the directed acyclic graph have completed. With this formulation, the client latency can be reduced significantly.

This transformation normally does not change the amount of work that must be done, it just improves the scheduling of the work. However, if a prefetch is issued for a query that is conditionally executed, the result of the prefetch will not always be used. While issuing such "speculative" prefetches increases the total work in the system, it allows trading off reduced latency for extra work. For our evaluation, we issued speculative prefetches whenever possible because more often than not, the result of the prefetches would be useful. The prefetches were wasted only when an error occurred in the execution of a query – an infrequent occurrence for the applications we studied.

In the technical report [6], we outline an algorithm for applying this transformation automatically and discuss implementation issues relating to this transformation.

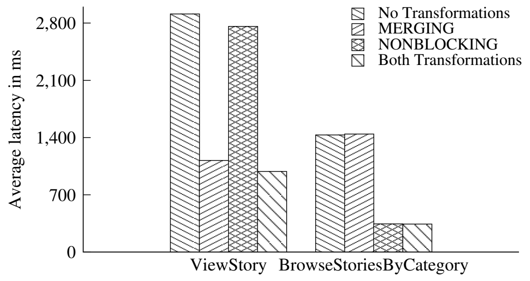


**Figure 4: Impact of the MERGING and NONBLOCKING transformations on latency. We show the average latency for two dynamic interactions in the BBOARD benchmark. The graph shows that the MERGING transformation has a significant impact on the average latency.**

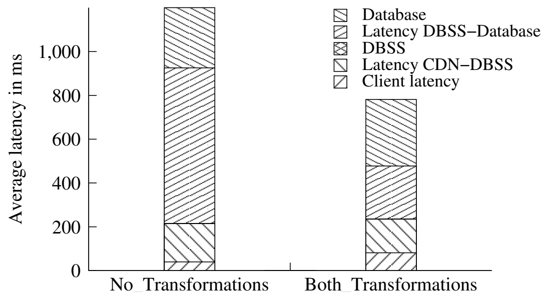


**Figure 5: Impact of the two transformations on the average latency of a dynamic interaction in the BBOARD application, executing in a DBSS setting.**

## 4. Evaluation

Due to space constraints, we provide only partial results here (detailed results are in [6]). We evaluate the two transformations–MERGING and NONBLOCKING–by applying them to three publicly available benchmark applications, which we refer to as AUCTION, BBOARD, and BOOKSTORE. In Section 4.1 we evaluate the effects of these transformations on latency, both in the traditional centralized setting as well as the DBSS setting. In Section 4.2 we list the frequencies with which the two transformations apply to our benchmark applications. We defer detailed "coverage" results of the transformations and their impact on "scalability" to [6].

### 4.1 Latency Impact of the Transformations

For our experiments, we used the setup of Figure 1(b). The latency between the client and the CDN node was 5ms, between the CDN node and the DBSS node was 5ms, and between the DBSS node and the home server was 100ms.

Figure 5 evaluates the impact of the two transformations on the average latency of a dynamic interaction in the BBOARD application, executing in a DBSS setting. (We chose BBOARD because the latency effects of the transformations on BBOARD is the highest.) The latency consists of five components: the client latency including the execution time at the CDN, the network latency from the CDN to the DBSS, the time spent

**Table 1: Runtime HTTP interactions in which the MERGING and NONBLOCKING transformation apply. The "either" column represents interactions in which at least one of the two transformations apply. The "static" column represents interactions in which a static HTML file is returned (clearly, neither transformation can apply to such interactions).**

| | % of runtime HTTP interactions | | | |
|---|---|---|---|---|
| Application | Static | MERGING applied | NONBLOCKING applied | Either |
| AUCTION | 15.9% | 15.2% | 3.8% | 15.7% |
| BBOARD | 7.4% | 69.8% | 28.5% | 70.1% |
| BOOKSTORE | 0.0% | 0.8% | 58.6% | 59.4% |

at the DBSS, the latency from the DBSS to the database, and the time spent at the database. Almost all the latency decrease is due to a reduction in the network latency from the CDN to the DBSS.

Figure 4 shows the effect of the transformations on the average latency, for two popular interactions in the BBOARD application, executing in a DBSS setting. Applying both transformations reduces latency by over 50%. Of the two transformations, the MERGING transformation causes a greater reduction in latency. The two transformations: MERGING and NONBLOCKING, are complementary. While the MERGING transformation can be applied only when the queries are *related*, the NONBLOCKING transformation can be applied only when the queries are *not related*. Consequently, we expect that applying both transformations results in the lowest latency, a hypothesis Figure 4 confirms.

## 4.2 Applicability of the Transformations

Table 1 lists the percentage of runtime HTTP interactions in which these transformations apply. The "either" column represents interactions in which at least one of the two transformations apply. The "static" column represents interactions in which a static HTML page is returned. Clearly, neither transformation can apply to such interactions. Even after including the static interactions (interactions which return an HTML file), one of these transformations applied to over 15%, 70%, and 59% of all runtime HTTP interactions for the AUCTION, BBOARD, and the BOOKSTORE benchmarks, respectively. For the BBOARD application, the MERGING transformation applies to over 69% of all HTTP interactions; this high percentage is one of the reasons why the MERGING transformation is particularly effective in reducing latency (Figure 4) of the BBOARD application. A similar argument can be made for the NONBLOCKING transformation and the BOOKSTORE application.

## 5. Summary

To meet their scalability needs, Web applications increasingly use a distributed server infrastructure. Inevitably, the network latency between the application and database servers increases in such settings. Two examples of such settings are: (i) a DBSS setting [5, 7] where different third party services may manage the application server(s) and the database server(s), (ii) a shared Web-service hosting scenario where the application and the database server typically run on separate clusters of machines, and typical latencies are between 16ms and 20ms [10]. Since a single HTTP request of a dynamic Web application typically results in multiple database queries, even a slight increase in the latency between the application and database server(s) increases the client latency significantly. In this work we proposed two holistic transformations—MERGING and NONBLOCKING—which can be implemented in a source-to-source compiler [4, 8]. These transformations reduce the latency by either clustering related queries or overlapping query execution. By manually inspecting our application code, we found opportunities to apply these transformations in 18.7%, 75.7%, and 59.4% of all dynamic interactions at runtime for the AUCTION, BBOARD, and the BOOKSTORE, respectively. These two transformations will continue to be useful as the two trends— using distributed infrastructures and issuing more database requests per HTTP requests—continue.

## References

[1] Akamai Technologies Inc. and Jupiter Research Inc. Akamai and Jupiter Research identify '4 seconds' as the new threshold of acceptability for retail web page response times. `http://www.akamai.com/html/about/press/releases/2006/press_110606.html`.

[2] Akamai Technologies Inc. and Quocirca. Akamai and Quocirca identify '4 second' performance threshold for European web-based enterprise applications. `http://www.edgejava.net/html/about/press/releases/2007/press_110707.html`.

[3] Hibernate. Relational persistence for Java and .NET. `http://www.hibernate.org`.

[4] L. J. Hendren et al. Soot: a Java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[5] A. Manjhi, A. Ailamaki, B. M. Maggs, T. C. Mowry, C. Olston, and A. Tomasic. Simultaneous scalability and security for data-intensive web applications. In *Proc. SIGMOD*, 2006.

[6] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic query transformations for dynamic web applications. Technical Report CMU-CS-08-160, Carnegie Mellon University, 2008.

[7] A. Manjhi, P. B. Gibbons, A. Ailamaki, C. Garrod, B. M. Maggs, T. C. Mowry, C. Olston, and A. Tomasic. Invalidation clues for database scalability services. In *Proc. ICDE*, 2007.

[8] ObjectWeb Consortium. ASM. `http://asm.objectweb.org`.

[9] Ruby on Rails. Active Records. `http://www.rubyonrails.org`.

[10] Simple measurements on the infrastructure of Dreamhost, a leading Web-hosting company. `http://www.dreamhost.com/`.