Transparent Caching with Strong Consistency in Dynamic Content Web Sites

Cristiana Amza
Department of Electrical and Computer Engineering
Toronto, Canada
amza@eecg.toronto.edu

Gokul Soundararajan
Department of Electrical and Computer Engineering
Toronto, Canada
gokul@eecg.toronto.edu

Emmanuel Cecchet
INRIA
Rhone-Alpes, France
Emmanuel.Cecchet@inrialpes.fr

Abstract

We consider a cluster architecture in which dynamic content is generated by a database back-end and a collection of Web and application server front-ends. We study the effect of transparent query caching on the performance of such a cluster. Transparency requires that cached entries be invalidated as a result of writes. We start with a coarse-grain table-level automatic invalidation cache. Based on observed workload characteristics, we enhance the cache with the necessary dependency tracking and invalidations at the finer granularity of columns. Finally we reduce the miss penalty of invalidations through full and partial coverage of query results.
In terms of system design, a query cache may be located at the database back-end, on dedicated machines, on the front-ends, or on a combination thereof. This paper evaluates the tradeoffs of the different cache designs and the cache location using the TPC-W benchmark.

Our experiments show that our transparent query cache improves performance very substantially by up to a factor of 1.5 in throughput and 4.2 in response time overall compared to the baseline table-based invalidation scheme. An important contributor to this end result, our optimization for reducing the miss penalty through full and partial coverage detection of query results from the cache improves response time by up to a factor of 2.9 compared to a cache with fine-grained column-based invalidations alone. Thus, the benefits of the higher hit ratio in our optimizations outweigh the costs of additional processing. The results are less clear-cut in terms of where to locate the cache. Performance differences when varying the cache location and the number of caches are small.

1 Introduction

A dynamic content web server generates responses to user requests by a combination of a web server, application logic, and a database (see figure 1). The web server receives the request, and causes the appropriate application logic to be executed. The application in turn sends a number of queries to the database, and constructs a reply based on the results of those queries. The web server then returns that reply to the requester. For the purpose of this paper we consider the combination of the web server and the application logic collectively as the front-end and the database as the back-end.

Database query caching has been proposed as a means for speeding up dynamic content generation [5, 8, 10, 12, 13, 18, 21]. The results of recent queries are cached, and re-used on subsequent queries, improving latency and reducing load on the back-end. If the back-end is the bottleneck, query caching also improves throughput of the overall server. Caching dynamic content is more complex than caching static content, because the cached entries may become invalid as a result of database writes.

Most of the earlier designs for query caching in dynamic content servers require extensive manual intervention [5, 9]. In these existing solutions a programmer or site administrator explicitly specifies the cacheable fragments of web pages and their lifetime. Other query caching solutions assume relaxed consistency semantics [2, 12] forcing the user to handle inconsistent results. Motivated by the strong consistency requirements of e-commerce workloads, we study their common workload characteristics with the goal to design a fully transparent query cache providing both performance enhancements and strict consistency guarantees to the user. We start from a basic transparent caching technique which provides strong consistency, but uses coarse-grained table-based invalidations, currently present as a feature inside our database [14]. We implement a similar cache with table-based invalidations, but outside the database for flexibility and portability, then we add a couple of optimizations based on observed workload characteristics.

First, we observe that in compute-intensive queries that involve table joins and sorting or grouping, only a couple of salient attributes are of interest to the user. These few selected attributes are disjoint from the set of database attributes that are typically updated in the same tables. Hence, we introduce caching with column-based invalidations that invalidates at the granularity of database attributes.

Next, we reduce the impact of automatic invalidations by either making them unnecessary or reducing the penalty of a cache miss through full or partial coverage of query results. Full or partial coverage occurs when a query response is fully or partially contained within one or more cached responses. We call these optimizations *semantic caching* because we use per-query knowledge to check for containment of the current query response within an already cached response.

Furthermore, we maximize the probability of significant partial coverage for read-only queries from the cache, by keeping track

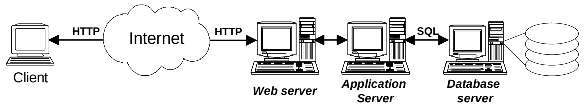

Figure 1. Common Architecture for Web Sites Serving Dynamic Content

of newly inserted rows in separate small temporary tables. A query result is then obtained from merging an existing cached response with one or more lightweight residual query results that may need to be computed on the temporary tables. By keeping the temporary tables small, the overall perceived response time for a query is low.

We evaluate the tradeoffs between the different cache designs and the system's architectural trade-offs between the various locations at which such cache(s) can be positioned. To transparently maintain consistency, the cache maintains, for each cached entry, a "dependency set" of database data items on which the cached entry depends. If one of the underlying database items changes, then the cached entry may no longer be valid. The underlying database data items may be specified at the granularity of database tables or database columns. Invalidation at the granularity of a table incurs little overhead when the dependency set is computed but triggers a large number of invalidations. Invalidation at the column level requires more elaborate parsing of the query to determine the dependency set, but reduces the number of invalidations. Similarly, our semantic cache optimizations can reduce the miss penalty through providing a full or partial response to the query from the cache. On the down side, these optimizations come at the expense of additional processing for containment checking, filtering responses for useless rows in cases of partial coverage and even reimplementing a limited fraction of the database functionality, such as, re-ordering, or re-counting rows during merging results.

The second tradeoff explored in this paper revolves around the location of the cache. The cache can either be co-located with the front-end or the back-end or it can reside on a separate machine. The more "upfront" the cache, the shorter the latency on a cache hit under low load. Throughput may, however, suffer if the front-end is the bottleneck. This tradeoff becomes more interesting if there are multiple front-end machines. If a cache is located on each front-end machine, then a cache only sees the traffic going through it, resulting in lower hit rates and consistency between the caches becomes an issue. Another alternative is to use a two-level cache, the first level resident at and private to each front-end, and the second level resident on a separate machine or on the database, and shared by all front-ends.

We have implemented a transparent query result cache that can use various granularities of invalidation and semantic cache enhancements based on partial query result coverage. The cache can be located at various locations in a dynamic content Web server. We have implemented our cache in various dynamic content servers running on common freeware software platforms. We have versions of the cache running with the Apache web server [1], the MySQL [14] relational database, and either PHP [17] or Java servlets for the application server. We use TPC-W [20] with the standard browsing and shopping mixes as our benchmark application. It models an online bookstore, such as amazon.com. We evaluate the tradeoffs discussed above by measurements of the throughput, the response time, and the hit rate in this implementation.

Our conclusions are as follows:

- Transparent query caching provides very substantial benefits in throughput and response time improvement respectively, without any changes to the Web server or the database, and without any hand-tuning of the application logic.
- Fine-grain invalidation is essential. The benefits of a cache with coarse-grain table invalidation similar to the one available in our base MySQL platform are much smaller.
- Our semantic cache extensions for partial coverage of query results adds up to a factor of 2.9 improvement to response time.
- The location of the cache has less impact on performance. The preferred strategy in the case of multiple caches depends on the workload. With a heavily read-dominated workload, a two-level cache, on the web servers and on a dedicated machine, is preferable. With more writes, a single shared cache on a dedicated machine performs best.

The outline of the rest of the paper is as follows. Section 2 describes the design of the cache, focusing on the different invalidation algorithms. Section 3 describes our enhancements for transparent partial query result coverage from the cache. Section 4 discusses the tradeoffs between different locations of the cache and the consistency protocol used in the case of multiple caches. Section 5 describes the experimental environment. Section 6 presents the results. Section 7 discusses related work. Section 8 concludes the paper.

2 Cache design

2.1 Overview

Logically, our dynamic content Web server consists of a front-end, containing both the web server and the application logic, a cache, and a database back-end. The cache functions as a transparent proxy between the application logic and the database: to the application logic, the cache appears as the database; to the database it appears as the application. The cache takes as its input the database queries generated by the application logic. On a read query, the cache checks whether the results of the query reside in the cache, and, if so, returns them immediately to the application. Otherwise, it forwards the query to the database. The database returns the results of the query to the cache, where they are inserted in the cache and forwarded to the application. On an update query, the cache performs the necessary invalidations and forwards the update to the database. The cache may be located on the same machine as the front-end, on a separate machine, or on the same machine as the database. There may be multiple machines executing the web server and the application logic. There may also be multiple cache machines.

2.2 Data structures

Figure 2 depicts the main cache data structures. The primary structure is a large hash table containing cache entries. Each entry contains the SQL query string, the state of the entry, the query result, the query's dependencies, and some additional fields to implement the cache replacement algorithm. The SQL query string serves as

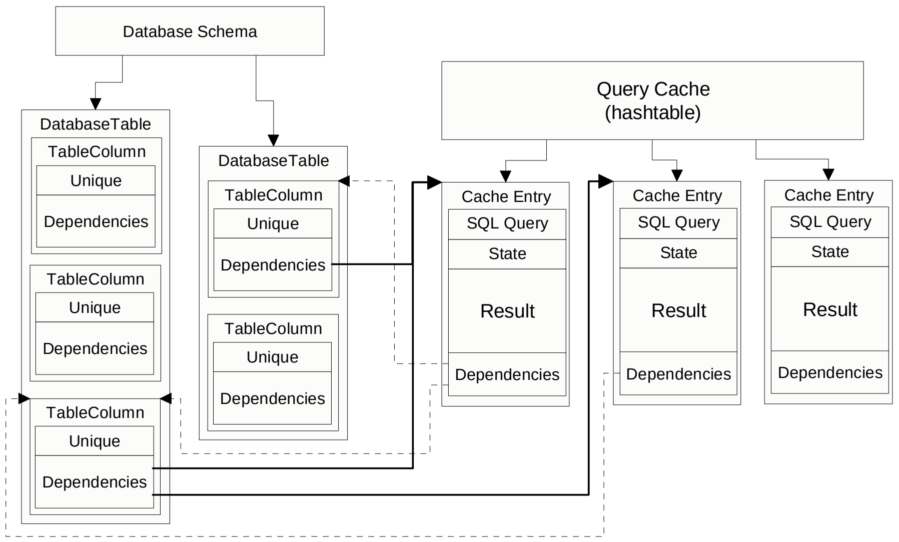

Figure 2. Cache Data Structures

the key to the hash table. A cache entry exists in one of the following states: valid, invalid, or single-row. The valid and invalid states correspond to their normal meanings. The single-row state reflects a cache entry that must contain exactly one row. It is used to implement an optimization to reduce invalidations described in Section 2.5. The query's dependencies record the data items in the database on which the result is dependent. For coarse-grain invalidation, these dependencies are recorded in terms of database tables, for fine-grain invalidation in terms of database columns. A second data structure records the schema of the database. It consists of a number of table objects, each one corresponding to a table in the database. In the case of coarse-grain invalidation, each table object contains references to the cache entries that are dependent on this table. In the case of fine-grain invalidation, each table object contains a number of column objects, each one corresponding to a column in the corresponding table. Each of these column objects contains references to the cache entries depending on this column. It additionally contains an indication of whether the column must contain a unique value. This field is used in the optimization described in Section 2.5.

2.3 Read query handling

If an incoming read query matches a valid entry in the cache, the cache simply returns the cached query results to the application. If there is no matching cached query, a new cache entry is allocated. The query is then forwarded to the database. While the database is performing the query, the cache parses the query string to determine its dependencies. In the case of coarse-grained cache invalidation, it parses the FROM clause in the query to determine the tables accessed. In the case of fine-grained invalidation, it parses the SELECT and WHERE clauses of the query to determine which columns of the tables named in the FROM field are used. In either case, appropriate references are inserted in the dependencies field of the cache entry, and in the references emanating from the database schema data structure. When the results of the query are received from the database, they are inserted in the cache entry, its state is marked valid, and the results are returned to the front-end.

2.4 Handling update, delete, and insert queries

When an update, insert or delete query is received, the query string is parsed to determine its dependencies, as in Section 2.3. Using the references in the database schema data structure, the cache invalidates all cache entries dependent on either the affected tables or the affected columns. These references are then deleted from the database schema data structure. The query is then forwarded to the database. When the response from the database is received, it is forwarded to the front-end.

2.5 Optimization for single-row cache entries

An insert can never invalidate a cache entry that represents a single row in the database. When parsing a read query, the cache can determine that a query must result in (at most) a single row being returned. A query returns at most a single row if the WHERE clause of the query contains a logical AND in which for each table named in the query there is an equality test of a unique column with a constant. Although there may be other circumstances under which no more than a single row can be returned, this test is conservative but easy to implement. When the cache discovers such a query, it sets the state of the cache entry to single-row rather than to valid. When an insert is processed, cache entries in single-row state are not invalidated.

2.6 Replacement policy

The cache implements an LRU replacement strategy by threading a doubly-linked list through the cache entries. The linked list maintains the cache entries in order of last access. When cache replacement is needed, the cache entry at the tail of the list is evicted.

3 Semantic Cache with Full and Partial Coverage of Query Results

Semantic caching is using per-query knowledge to check for containment of the current query response within an already cached response. For this purpose, we parse each query to infer query result ranges and possible coverage or partial coverage for all queries that match a specific template.

An orthogonal but related transparent cache optimization is based on addressing the invalidations caused by the addition of new items to a database table in a cache with table or column-based automatic invalidations.

We maximize the probability that a read-only query can be largely satisfied from the cache through partial coverage, by keeping track of newly inserted items in separate small temporary tables. A query result is then obtained from merging an existing cached response with one or more lightweight residual query results that may need to be computed on the temporary tables. By keeping the temporary tables small, the overall perceived response time for a query is low.

The combination of partitioning the new and old items through the use of temporary tables driven by application patterns and using per-query semantic information for detecting coverage work in synergy. In particular, partitioning increases opportunities for partial coverage for each individual query. We explain the table partitioning scheme, our method for detecting coverage of query results, filtering unwanted parts of responses and merging cached responses with residual query responses in more detail in the following sections.

3.1 Partitioning Scheme for Alleviating Insert Induced Invalidations

We keep newly inserted rows in separate tables called *temp tables*, one temp table per regular database table. The value of each field in the temp table rows (including key values) is the same as if the rows were present in the main table. Upon receiving a select query, the cache splits it into two queries: the original query and one or more residual queries. The original query is the unmodified query working on the regular table. A residual query is the same query on the corresponding temp_table. These separate results are obtained and cached as usual. The final query result is obtained by merging these query results. This optimization potentially benefits query results that would otherwise be invalidated by inserts into the accessed tables. In the example shown in Figure 4, a `SELECT` which fetches a large number of rows (A) would be invalidated by a subsequent `INSERT` query. By placing newly inserted rows in a temporary table, we avoid invalidating the result of the first `SELECT` and we can compute the result for a subsequent matching `SELECT` quickly by merging the cached response (A) with a small residual response (B) computed from the corresponding temporary table (Figure 5).

Although the residual queries on the temporary tables are lightweight, we choose to cache their results as well. Hence, these residual results could either be returned from the cache if valid or otherwise would need to be recomputed at the database similarly to the treatment of any other query.

Upon an `INSERT` query, the cache redirects the query to insert it in the temporary table. For `UPDATE` and `DELETE` queries the cache sends them to both the original and the temporary tables. In the case of queries containing joins of two or more tables, the query is split into the corresponding sub-queries necessary to compute the join according to the formula shown in Figure 3.

Partitioning a select query could lead to $O(2^n)$ partitioned queries, where n is the number of tables referenced in the query. However, in practice, only certain tables are partitioned at any given time (usually 1 or 2). Their number depends on how many of a query's tables registered inserts in the recent past. Furthermore, the joins that involve the temporary tables are fast since these tables contain a few rows (a maximum of 100 in our scheme). When an insert table exceeds the threshold size of 100 rows, we remove and reintegrate all entries from the insert table into the corresponding regular table. Since the number of sub-queries is kept low and the temp tables are small, computing the residual queries in the case of joins is relatively fast compared to recomputing the original result.

$$\begin{aligned} A \bowtie B &= (A \cup tA) \bowtie (B \cup tB) \\ &= (A \bowtie B) \cup (A \bowtie tB) \\ &\quad \cup (tA \bowtie B) \cup (tA \bowtie tB) \end{aligned}$$

Figure 3. Join Formula for Partitioned Tables (*shown for 2 tables*)

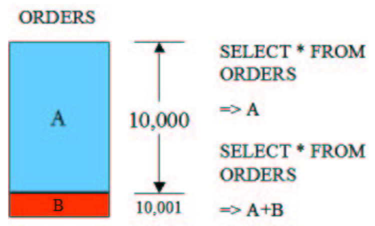

Figure 4. Incremental change for a query response (A) upon inserting a new row (B)

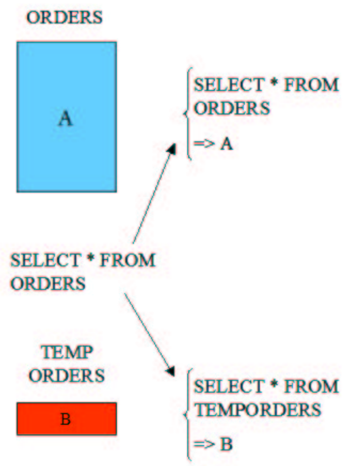

Figure 5. Using a separate temporary table for new inserts allows reuse of previous query response (A) for partial coverage

3.2 Detecting Coverage for Query Results

The cache can detect both *Full Coverage* and *Partial Coverage* of a query response. In *Full Coverage* (see Figure 6), the query response of the current query is fully contained within a cached response. In this case, (at least) one of the cached query responses is a superset of the current query response. In *Partial Coverage*, a query result is obtained from merging an existing cached response with one or more residual query results that may need to be computed at the database.

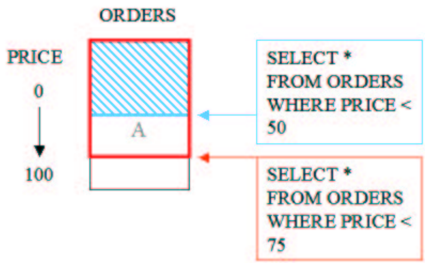

Figure 6. Coverage of Query Results: The response for the "PRICE $<$ 50" query is fully contained within a previous cached entry

Full coverage is classified as a cache hit, while partial coverage is currently classified as a cache miss. Checks for either full or partial coverage are done for all types of queries (including sub-queries of a partitioned query). The cache first checks whether there is an exact match with a previously cached query. If not, the algorithm proceeds to check if any previously cached query can provide a full answer (full coverage). If full coverage is detected, the cache filters out any rows that are not necessary to satisfy the current query from the cached entry and returns the resulting query response.

Otherwise, the cache algorithm proceeds to check if any previously cached query can provide a partial answer. If such a query exists, the cache sends a remainder query to the database. When the database returns the result for the remainder query, the cache merges the result with the partial result obtained from the cache. In the following section we provide more detail on how we determine whether a cached result satisfies the incoming query (i.e., how we implement full coverage and partial containment checks), how to generate a remainder query in the case of partial coverage and the merging algorithm.

3.2.1 Coverage Checking

We follow the general coverage testing described by Larson et al. [11] which defines that a query Q_1 *covers* Q_2 if the following three conditions hold:

1. *Attribute Coverage:* This states that the columns in the `SELECT` clause of Q_2 should be a subset of Q_1.
2. *Tuple Coverage:* This means that the tuples addressed by Q_2 should be a subset of the tuples addresses by Q_1. In other words, for the `WHERE` predicates, P_1 of Q_1 and P_2 of Q_2, $\forall (ttupleofQ_2) P_2 \rightarrow P_1$.
3. *Selectability:* This requires that the query must be entirely evaluated using the cache entries. That is, Q_2 should not refer to any columns in other `SQL` constructs (such as `ORDER BY`, and `WHERE`) that is referred by Q_1.

We currently consider only conjunctive `WHERE` clauses for containment (coverage) checking. Furthermore, we can detect coverage only conservatively for queries matching a given query template (e.g., differing only in the inequality expressions appearing in the `WHERE` clause).

In more detail, the containment checking algorithm takes as input the incoming query Q and a list of cached queries C matching Q's template. Each query contains a list of predicates P contained in the `WHERE` clause. For each cached query, the algorithm checks whether all the equality constraints are satisfied. Any cached query not satisfying an equality constraint is removed from further consideration for this match. Then, for each inequality predicate, the algorithm selects the cached query that *covers* the most tuples. In case of ties, the first query on the list is returned.

3.2.2 Generation of the Remainder Query

To complete the result obtained from the cache, a remainder query obtains the missing tuples from the database. Formally, the remainder query is described as $R = Q - C$ where Q is the incoming query and C is the cached partial result. The remainder query is generated in a straightforward way. We consider the inequality predicates, one at a time. For each predicate, if the cached query does not cover the incoming query, the predicate is rewritten to fetch the tuples not referred by the cached query.

For example, consider the following queries conforming to template T below, where the new query Q is presented to the cache, while the results of C1, C2 and C3 are already cached (only the `WHERE` clause is shown for these queries):

```
T: (SUBJECT = ?) AND (PRICE < ?)
Q: (SUBJECT = 'KIDS') AND (PRICE < 100)
C1: (SUBJECT = 'KIDS') AND (PRICE < 50)
C2: (SUBJECT = 'KIDS') AND (PRICE < 75)
C3: (SUBJECT = 'ARTS') AND (PRICE < 100)
```

The containment checking algorithm first checks whether the equality constraints are satisfied. After this stage, `C3` can be removed from future consideration. Then, we proceed to check if there are any overlapping regions between Q and any of the cached queries. In this example, it is easy to see that both `C1` and `C2` partially cover `Q`, but `C2` is a better candidate since it contains a larger percentage of the final result. Therefore, `C2` is selected as the best result from the cache. `Q` is rewritten to

```
Q2: (SUBJECT = 'KIDS') AND
        (PRICE >= 75 AND PRICE < 100)
```

The rewritten query is sent to the database and its results are merged with the cached results to form the final answer that is returned to the client.

3.2.3 Efficient Merging

Since results have to be merged for every partitioned query, special care needs to be taken for queries that contain special clauses such as `ORDER BY`, `COUNT`, `MAX` and other SQL functions. For sorting necessary in `ORDER BY` clauses, we use an algorithm that merges the results in $O(ndc)$ time where d is the number of partitioned tables in the query and c is the number of `ORDER BY` columns. Our algorithm is based on merge-sort [7] and uses the fact that the query result sets to be merged are pre-sorted by the database. The algorithm then generates the sorted final result by interweaving the different streams.

4 Location of the cache(s)

The second tradeoff explored in this paper revolves around the location of the cache. The cache can either be co-located with the front-end or the back-end or it can reside on a separate machine. The more "upfront" the cache, the shorter the latency on a cache hit under low load. Throughput may, however, suffer if the front-end is the bottleneck. This tradeoff becomes more interesting if there are multiple front-end machines. If a cache is located on each front-end machine, then a cache only sees the traffic going through it, resulting in lower hit rates and consistency between the caches becomes an issue.

4.1 Consistency between multiple front-end caches

Consider a scenario with one cache per front-end machine. There is no dedicated cache machine, and there is no cache on the back-end machine. Consistency is maintained between the different caches using an invalidation protocol implemented by reliable group communication. When an update, insert or delete query arrives at a particular front-end, it is first forwarded to the back-end. The query is then parsed as before to determine its dependencies. An invalidation message containing these dependencies is sent to all the other caches by means of reliable group communication. The front-end to which the query was initially sent then waits for the multicast to be acknowledged by all the other caches and for the response from the database to come back. At that point, it sends the response back to the client. In this way, the invalidations are usually overlapped in time with back-end access and do not contribute to the latency seen by the client. They do, however, still contribute to the CPU load on the front-ends. In addition, each cache only sees the read traffic arriving locally, thereby limiting its hit rate. The next section describes a two-level cache that addresses this problem.

4.2 Two-level cache

In this scenario, there is a cache at each individual front-end, as before, but, in addition, there is a single shared cache, either on a dedicated cache machine or on the database. An incoming read query is first checked for a match in the cache of the front-end at which it is received, and served from there in the case of a hit. In the case of a miss, a cache entry is allocated for it, and the query is forwarded to the shared cache. The second-level cache behaves identically to the first-level cache. On a hit, it returns the cached results to the first-level cache. On a miss, it allocates a cache entry and forwards the query to the back-end. The returned results are then stored in the caches as necessary. An incoming update, insert or delete query is handled as before. The receiving front-end forwards it to the shared cache, and then initiates the group invalidation protocol among the front-ends. The shared cache forwards the query to the back-end, and performs its own invalidations. The back-end response is forwarded, when all the necessary invalidations have been completed.

5 Experimental Platform

5.1 Hardware Platform

We use the same hardware for all machines running the client emulator, the Web servers, the cache and the database. Each machine has an AMD Athlon 800Mhz processor, 256MB SDRAM, and a 30G ATA-66 disk drive. All machines are connected through a switched 100Mbps Ethernet LAN. We have verified that the amount of memory on the machines, the disks, and the network never become the bottleneck.

5.2 Software Environment

All machines run FreeBSD 4.1. We use Apache v.1.3.22 as our Web server, configured with the PHP v.4.0.1 module, providing server-side scripting for generating dynamic content. We use MySQL v.3.23.43-max as our database server. We increase the maximum number of Apache processes to 512. With that value, the number of Apache processes is never a limit on performance.

5.3 TPC-W Benchmark

We use an industry-standard e-commerce benchmark, TPC-W, from the Transaction Processing Council [20]. Several interactions are used to simulate the activity of a retail store. We implemented the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 scripts, 6 are read-only, while 8 cause the database to be updated. The read-only interactions include access to the home page, listing of new products and best sellers, requests for product detail, and two interactions involving searches.

TPC-W specifies three different workload mixes, differing in the ratio of read-only to read-write interactions. The browsing mix contains 95% read-only interactions, the shopping mix 80%, and the ordering mix 50%.

The database size is determined by the number of items in the inventory and the size of the customer population. We use 10,000 items and 2.8 million customers which results in a database of about 4 GB.

5.4 Measurement Methodology

We use the browsing and shopping workloads of TPC-W to investigate the various caching trade-offs. We use a client-browser emulator that allows us to vary the load on the web site by varying the number of emulated clients. To select the load for each experiment, we are driving the server without the cache with increasing the number of clients, until performance peaks. Then we use the same number of clients to drive the server with the cache enabled, for all cache locations and invalidation granularies. Each experiment with a particular workload mix is run for one hour, where the first half of the run is used for warming up the cache, and measurements are performed during the second half of the run. Each experiment also starts with an identical database. Differences between repeated runs of the same experiment were minimal. To measure resource utilization we use the vmstat utility that collects CPU, memory, and disk usage every second.

6 Results

We study the application access patterns relevant to scaling optimizations for a transparent caching approach for e-commerce using the TPC-W benchmark in section 6.1. Then we study the impact of our various cache optimizations compared to a query cache with table-based automatic invalidations.

6.1 Preliminary Results: Workload Characteristics Relevant to Caching

In the following, we describe the e-commerce workload characteristics that we found relevant for query caching. Our design is driven by the features below.

Heavyweight Database Processing: Table 1 shows the breakdown of the average total client response time for all TPC-W interactions into percentage of total response time spent at the Database, in Application server processing and in Web Server processing for the

Mix	DB	App Server	Web Server
Browsing	95%	4.0%	1.0%
Shopping	94%	4.4%	1.5%

Table 1. Breakdown of response time into percentage of total response time spent at the Database, in Application Server processing and in Web Server processing for the Browsing and Shopping mixes

two TPC-W workload mixes. We see that most of the time is spent in processing queries at the database, resulting in high potential performance impact for database query result caching.

Workload Locality: This is a motivator for dynamic content caching in general. The queries are repeatable and conform to predefined query templates. All read-only interactions exhibit locality in their access patterns, which ranges from hot-spot rows satisfying a condition (such as top-k published items and top-k best Sellers), to larger sets of frequently accessed rows in the item and customer tables for bestseller, new product and promotional items, and return customers respectively. The measured total size of all possible responses for all complex repeatable queries (BestSellers, NewProducts, Search by Subject) across all subjects was less than 300 KB. Hence, the read queries consuming the most database resources can be satisfied with a low total cache size.

Few Interesting Attributes for Heavyweight Browse Queries: This characteristic encourages use of a transparent query cache with automatic column-based invalidations. A column-based cache avoids performing cache invalidations as a result of updates in this case, as explained below. Complex read-only queries access only a few interesting attributes out of a high total number of attributes for the accessed tables. Specifically, a BestSeller query selects only the book title and author name ignoring any other attributes from the book and author tables such as the author biography, the book stock and the publication date. The interesting attribute set for these browsing queries is different from the attribute set accessed in update queries for the same tables. For instance, a book order will update a book's stock, but the book stock is typically not accessed in BestSeller, Newproduct and Search by Subject queries. Thus, a cache organization that distinguishes objects at the level of a database attribute (column) will not invalidate the results of the complex join queries as a result of updates occurring on the same tables.

One-shot Browse Queries: As opposed to the database hogs above, one-shot database queries involved in Product Detail or Customer Information type queries typically select all book or customer attributes respectively. However, only one such row is selected, typically based on the unique key of the particular row (e.g., Customer Info for customer with ID = 1000). These type of query results should not be affected by inserts of new rows into the respective tables such as adding new customers to the store's database.

High Fraction of Inserts: Of all write queries occurring in the standard shopping workload mix, 52% are inserts, 37.4% are updates and 11% are deletes. Furthermore, we observe that in dynamic content web applications, the most recently inserted rows will also be typically accessed in complex browse-type queries. For example, the newest orders for books in a bookstore influences BestSeller rankings for the books in the corresponding categories. Hence, as opposed to updates, inserts will typically invalidate the results of the database hogs (BestSellers, NewProducts, Searches) causing hefty cache miss penalties.

Method	WIPS	RespT	Hit %
NoCache	10.51	2.72	
Table	13.50	1.84	11.10
Column	19.14	0.98	50.40
Column + single-row	18.81	1.00	50.50

Table 2. Throughput in web interactions per second, response time, and hit rate for browsing mix with dedicated cache machine

Method	WIPS	RespT	Hit %
NoCache	18.71	1.05	
Table	19.51	0.94	4.40
Column	22.57	0.64	37.20
Column + single-row	22.99	0.61	38.20

Table 3. Throughput in web interactions per second, response time, and hit rate for shopping mix with dedicated cache machine

Subject-based Complex Queries: Most complex read-only queries are subject-based. They are either top-k type of queries by category (e.g., top 50 romantic BestSellers) or searches by category (e.g., the titles and authors of books with subject 'ARTS'). Hence, semantic caching that understands regions in the query result space is likely to improve the transparent query cache effectiveness.

In the following, we evaluate the performance impact of our cache optimizations and the cache location. In Section 6.2 we consider the scenario with a single cache located on a dedicated machine, and we vary the invalidation granularity.

In Section 6.3 we stay with a single cache, we pick the best invalidation granularity from Section 6.2, and we vary the location of the cache. In Section 6.4 we consider the case of multiple caches.

In Section 6.5 we consider the best location of the cache scenario and we enhance the cache with full and partial coverage of query results through table partitioning, coverage detection and merging of results.

6.2 Single cache - varying the invalidation granularity

This section reports on the results of a set of experiments with a single cache on a dedicated machine, while varying the cache granularity. The experiments use either no caching or caching with coarse-grain invalidation, with fine-grain invalidation, and with fine-grain invalidation with the optimization for single-row cache entries. Table 2 provides throughput, latency and hit rate, for the browsing mix, and Table 3 provides the same information for the shopping mix.

No matter what invalidation strategy is used, caching improves both

Mix	Q res.	Query	Meta	Total
Browsing	11.27	3.52	13.40	28.19
Shopping	12.69	4.39	15.93	33.01

Table 4. Average cache size for the browsing and shopping mixes with column based invalidations and single row optimization with dedicated cache machine. The total data stored in the cache is broken down into the average storage used for query results, query strings and cache metadata, respectively. All sizes are in MB.

Location	Throughput	Response Time
Database	18.91	1.01
Dedicated	19.14	0.98
Web server	19.14	0.90

Table 5. Throughput and response time for various locations of the cache with the browsing mix and column-based invalidation

Location	Throughput	Response Time
Database	22.47	0.64
Dedicated	22.99	0.61
Web server	21.43	0.78

Table 6. Throughput and response time for various locations of the cache with the shopping mix and column-based invalidation

throughput and response time for both workload mixes. Fine-grain column-based invalidation substantially outperforms coarse-grain table-based invalidation. The gain in hit rate is very substantial, and outweighs the small extra cost in more complicated parsing. Hit rates improve from 11.1% to 50.4% for the browsing mix, and from 4.4% to 37.2% for the shopping mix. There is a larger fraction of writes in the shopping mix, which explains both the lower overall hit rate and the greater difference between column-based and table-based invalidation. Average response time for the browsing mix improves from 2.72 seconds without a cache, to 1.84 seconds with table-based invalidation and 0.98 seconds with column-based invalidation. For the shopping mix, very little improvement results from using a cache with table-based invalidation, from 1.05 to 0.94 seconds, because the hit rate is very low. The much higher rates of the cache with column-based invalidation lead to a significantly improved average response time of 0.64 seconds. Since the database becomes the bottleneck during some periods of the experiment, the cache improves throughput, in addition to producing better response times. For the browsing mix, throughput increases from 10.51 interactions per second in the absence of a cache, to 13.50 with table-based invalidation, and 19.14 with column-based invalidation. For the shopping mix, the corresponding throughputs are 18.71, 19.51, and 22.57. Again, throughput improvement is small for table-based invalidation with the shopping mix, because the hit rates are small. With column-based invalidation, the improvements are significant for both mixes. The single-row optimization improves the performance of fine-grain invalidation by an insignificant amount. The increases in hit rates are small (from 50.4% to 50.5% for browsing and from 37.2% to 38.2% for shopping). In subsequent performance results, we only present results for the column-based invalidation strategy without the single-row optimization.

We also present measurements of the size of the cache for the best strategy in table 4. We can see that all cache sizes were small totaling less than 40 MB.

6.3 Single cache - varying the cache location

The relatively small cache sizes and the small CPU load on the dedicated cache machine, observed in the experiments described in Section 6.2, motivate an investigation of the effect of locating the cache on the front-end or on the back-end machine. Table 5 shows the throughput and the response time with the cache on the (single) front-end, on a dedicated machine (as in Section 6.2), or on the database for the browsing mix. Table 5 shows the corresponding results for the shopping mix.

The results show that depending on the other possible locations of the cache, the results differ only by small amounts. We have also established that our cache with table-based invalidations, when placed at the database, has very similar performance results and hit rates to enabling the MySQL internal cache alone (not shown). For the browsing mix, the database is the bottleneck during substantial periods of the execution. Compared to a cache on a dedicated machine, putting the cache on the database therefore reduces throughput and increases response time, both by a small amount, from 19.14 interactions per second to 18.91 for the throughput, and from 0.98 seconds to 1.01 for the response time. In contrast, moving the cache to the front-end improves response time, again by a small amount, from 0.98 to 0.90 seconds, because of one fewer network round-trip in the case of a hit. For the shopping mix, the database and the front-end alternate in being the bottleneck during the execution. Therefore, putting the cache on either machine slightly degrades performance compared to the dedicated cache. Throughput goes down from 22.99 interactions per second on a dedicated machine, to 22.47 on the database and 21.43 on the front-end. Response time goes up from 0.61 seconds to 0.64 and 0.78 seconds, for the database and the front-end, respectively. Summarizing these results, the low overhead of the cache makes performance relatively insensitive to the placement of the cache.

Location	WIPS	RespT	Hit %
Dedicated	20.79	0.86	51.00
Web server	16.66	1.35	26.90
Two-level	20.80	0.80	29.70 + 29.40

Table 7. Throughput, response time, and hit rate for the browsing mix for 4 Web servers, with a single shared cache, a private cache on each Web server, and a two-level cache consisting of a private cache on each front-end and a shared cache on a dedicated machine

Location	WIPS	RespT	Hit %
Dedicated	27.77	0.39	40.30
Web server	24.96	0.55	21.10
Two-level	26.88	0.42	21.10 + 29.60

Table 8. Throughput, response time, and hit rate for the shopping mix for 4 Web servers, with a single shared cache, a private cache on each Web server, and a two-level cache consisting of a private cache on each front-end and a shared cache on a dedicated machine

6.4 Multiple caches

With a single front-end, as in Section 6.3, it appears that small gains are possible by locating the cache on the front-end versus locating it elsewhere. This result has to be re-examined when there are multiple front-ends, because of the need to enforce consistency between the front-end caches and because of the fact that each front-end cache only sees the traffic going through its front-end. Table 7 shows throughput, response time and hit rate for the browsing mix for a single shared cache on a dedicated machine, a private cache on each of the front-ends, and a two-level cache consisting of a private cache on each front-end and a shared cache on a dedicated machine. Table 8 reports the corresponding results for the shopping mix.

First, putting only a private cache on each Web server does not perform as well as a single shared cache on a dedicated machine. The hit rates decrease because each web server only sees a fraction of the overall traffic compared to the shared cache. Both throughput and response time deteriorate as a result. For the two-level cache with private caches on the Web servers and a shared cache on a dedicated machine, the results differ between the browsing mix and the shopping mix. The differences result from the differing fractions of writes in the workloads. The shopping mix has four times more writes than the browsing mix, resulting in four times more consistency messages. The cost of these messages causes a slight reduction in performance from a single cache on a dedicated machine to a two-level cache. For the browsing mix, in contrast, a small improvement results from a two-level cache. This result merits some further discussion. First, an application that puts a heavier load on the front-end may produce different results. Regardless of caching, the benefits of adding extra front-ends for such an application should be larger than for TPC-W, which puts a heavy load on the database. Second, our implementation of caching on the front-end resides in a separate process from the Web server, so that the cache can be shared between the different Apache processes. A more integrated solution may produce better results. Finally, the current implementation of consistency communication uses TCP. A more efficient communication mechanism may alter the tradeoff.

6.5 Semantic Caching Results

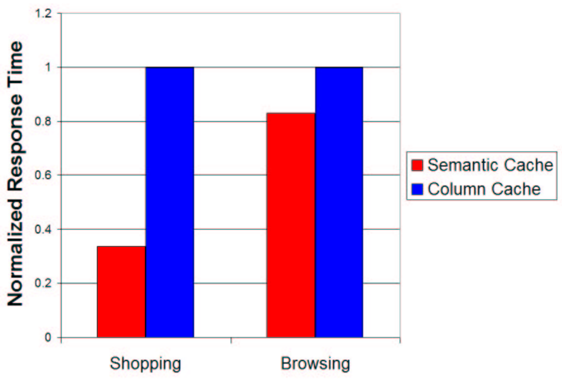

Figure 7. Normalized latency comparison for column-based cache and semantic cache, normalized to the column-based cache.

Figure 7 shows a latency comparison for our cache with column-based invalidation and the semantic cache for the TPC-W browsing and shopping mixes, respectively. The latency represents the average response time for a page as perceived by the client normalized to the response time with the column-based cache. The results indicate that semantic caching further lowers the latency significantly, by a factor of 2.9 for the shopping mix and by a factor of 1.2 for the browsing mix. The shopping mix contains a higher fraction of writes compared to the browsing mix (20% vs. 5%), hence a higher number of automatic invalidations. The latency gains are mostly due to faster computation of query responses from partial and residual results compared to re-executing the original query.

Semantic caching also improves the hit-rate of the cache due to full coverage detection. For the browsing mix, the hit-rate is improved by 8% and for the shopping mix by 4%. These increases in hit rate are not significant because they result only from additional cases of full coverage of query results and servicing these directly from the cache. We currently classify partial coverage as misses.

Thus, most of the latency improvements come from partial coverage cases rather than additional hits due to full coverage. Partial hits, especially for queries that involve multiple table joins (e.g., the *best seller* query that retrieves the books that were ordered the most in recent purchases) make a difference in terms of latency improvements for the shopping mix. The performance improvements brought about by semantic caching translate in throughput increases as well, however to a lesser extent than for latency, with throughput increases only up to 10% for both mixes.

7 Related Work

7.1 Overview of dynamic data caching

Dynamic Web data can be cached at different stages in its production: the final HTML page (e.g., [4, 10], intermediate HTML or XML fragments (e.g., [8]), database queries (e.g., [13]), or database tables (e.g., [12, 15]). Combination of various caches are also possible (e.g., [5, 21]). Intuitively, caching at the database stage typically offers higher hit ratios, while caching at the HTML or XML stage offers greater benefits in the case of a hit. There is no conclusive evidence at this point that caching at any single stage dominates the others. For instance, Labrinidis and Roussoulos use a synthetic workload and conclude that HTML page caching is superior [10], but Yagoub et al. use TPC-D and conclude that database query caching is more effective [21]. It appears that the different caches are complimentary [18, 21]. This paper is concerned with database query caching. Our methods can be extended to record dependencies between HTML pages or fragments and database data items, and we intend to investigate this in further work.

7.2 Non-transparent approaches

Luo et al. [12] require the database designer to specify which tables are cached. Updates to the cache are performed once a minute. Oracle 9i also provides table-level caching in the middle-tier and invalidation based on time and events (database triggers), but no generalizable solution for generating invalidations [15]. Yagoub et al. [21] describe a declarative system for specifying a web site that allows a designer control over HTML, XML and query caches, including what to insert or to remove from the cache and how to invalidate or update items in the cache. Challenger et al. propose a cache API to control the contents of the API [9, 5]. Datta et al. [8] propose annotating the application logic to inform the cache which HTML fragments are cacheable. In contrast, our approach is transparent, can be applied without additional effort to an existing web site design, and automatically maintains consistency at all times. Nonetheless, we have been able to demonstrate substantial performance benefits.

7.3 Caching in clusters of dynamic web servers

To the best of knowledge, this paper is the first in-depth evaluation of query caching in a cluster of web server front-ends. Our previous work [3] studies the ortogonal aspect of database clustering and combinations of database clustering and transparent query caching. Challenger et al. [5] use an analytical cost model to evaluate how many web servers can be supported by a single cache, based on the CPU cost of executing queries on one hand and maintaining the cache on the other hand. They describe no implementation or experimental evidence in support of their numbers.

7.4 Multiple caches

Our consistency maintenance algorithm is similar to the ones commonly used in small shared-memory systems. An alternative used in larger shared-memory systems is to use a directory-based rather than a multicast-based invalidation protocol. For configurations with larger number of front-ends, this is a plausible avenue for further exploration. Alternatively, one could also use weaker consistency semantics present in some shared-memory systems. We use a second-level shared cache to augment the hit rate with caches on multiple front-ends. This allows us to use a simple round-robin request distribution for directing incoming client requests. More complicated request distribution strategies such as LARD [16] can potentially further increase the hit rate in the front-end cache, but they need support from the switch. Cooperative caching is another approach to improve cache hit rates, but we prefer the simplicity of the shared cache.

7.5 Semantic Caching

The concept of semantic caching has been examined in the context of database design [11, 19]. This approach has been explored for LDAP (Lightweight Directory Access Protocol) with existing studies [6] focusing on how to reuse results from existing LDAP queries to answer future queries. More recently, Amiri et al. [2] has proposed using semantic information to generate results based on cached query results for dynamic content queries sharing the same query template. Their cache shares similarities with our per-query semantic information optimizations, but it lacks the ability to generate partial results. Furthermore, their intended deployment is in caches with loose consistency at the client edge of the network, while our focus is on providing strong consistency for a central server cache.

8 Conclusions

We have demonstrated that transparent dynamic caching substantially improves the performance of dynamic content Web servers. For the TPC-W benchmark, we have shown factors of up to 1.5 cumulative improvement in throughput and factors of up to 4.2 cumulative improvement in response time, compared to a basic table invalidation scheme without requiring changes to the Web server or the database, and without any hand-tuning of the application logic. In order to provide these substantial improvement, cache invalidations must occur at the grain of a database column. Very small additional benefits can be obtained from specializing the case of a single-row cache entry. We have also shown that a more coarse-grained table-based invalidation scheme currently used by MySQL, provides little improvement by itself, compared to an architecture with no cache because the resulting hit rates are too low.

As a further improvement we presented a method of using semantic information to retrieve partial results for queries from the cache. This method has several new features. First, the tables are partitioned to reduce misses due to `INSERT` queries. Second, information from previous queries is used to generate partial results.

The location of the cache appears less critical to performance. When there is only a single front-end, a dedicated machine is preferable, but the differences with other locations are small. With multiple front-ends, caches at the front-ends are appealing only in combination with a shared cache and when the write component of the workload is sufficiently small not to cause significant consistency overhead.

9 References

[1] The Apache Software Foundation. http://www.apache.org/.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking in web semantic caches. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, Bangalore, India, 2003.

[3] C. Amza, A. Cox, and W. Zwaenepoel. A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers. In *Proceedings of the 21st International Conference on Data Engineering*, April 2005.

[4] K. Selcuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, May 2001.

[5] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of IEEE INFOCOM'99*, pages 294–303, March 1999.

[6] Sophie Cluet, Olga Kapitskaia, and Divesh Srivastava. Using LDAP directory caches. pages 273–284, 1999.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to algorithms (second edition). McGraw-Hill and MIT Press.

[8] Anindya Datta, Kaushik Dutta, Helen M. Thomas, Debra E. VanderMeer, Krithi Ramamritham, and Dan Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 667–670, September 2001.

[9] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. December 1997.

[10] Alexandros Labrinidis and Nick Roussopoulos. WebView Materialization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 367–378, May 2000.

[11] P.-Ȧ. Larson and H. Z. Yang. Computing Queries from Derived Relations. In *Proceedings of the 11th International Conference on Very Large Databases*, pages 259–269, August 1985.

[12] Q. Luo, S. Krishnamurty, C. Mohan, H. Pirahesh, HW̃oo, B. Lindsay, and J. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM International Conference on Management of Data*, pages 600–611, June 2002.

[13] Qiong Luo and Jeffrey F. Naughton. Form-based proxy caching for database-backed web sites. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 667–670, September 2001.

[14] MySQL. http://www.mysql.com.

[15] Oracle. Oracle9*i* Application Server Web Caching, October 2000.

[16] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, October 1998.

[17] PHP Hypertext Preprocessor. http://www.php.net.

[18] Karthick Rajamany. *Multi-tier caching of dynamic content for database-driven web sites*. PhD thesis, Rice University, August 2000.

[19] Daniel J. Rosenkrantz and Harry B. Hunt III. Processing conjunctive predicates and queries. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, pages 64–72. IEEE Computer Society, 1980.

[20] Transaction Processing Council. http://www.tpc.org/.

[21] Khaled Yagoub, Daniel Florescu, Valerie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 188–199, September 2000.