A Cache Engine for E-Content Integration

Content-integration systems generally suffer performance bottlenecks due to network overhead. To address this problem, the authors developed the Data Integration Cache Engine (DICE), which uses summarization techniques (subqueries and Bloom filters) and semantic metadata to achieve semantic active caching in the context of Internet-based data integration applications. The system uses algorithms and specialized data structures to generate exact remainder queries to locate content that is missing from cache in case of partial hits. The authors' performance results indicate that DICE outperforms existing options in terms of response time, network overhead, and server load.

Peter Triantafillou and Nikos Ntarmos
University of Patras, Greece

John Yannakopoulos
University of Crete

The most widespread approach to automating data integration and dissemination over the Web is to employ *wrappers* — software components that extract information from back-end sites and translate it to a common model — and a *mediator*, which combines the data gathered by the wrappers and provides uniform access to it. Researchers generally focus on the wrapping aspect, but bottlenecks in the real world usually come from the network overhead incurred when contacting integrated e-commerce sites. Although wrapping can be a tedious task, it requires only processing horsepower, which is abundant in modern computer systems. On the other hand, network congestion and link failures are part of everyday life on the Internet.

Caching queries and their results within a data integration system provides an effective and transparent method for tackling the network overhead problem. To that end, we developed the Data Integration Cache Engine (DICE), which employs *semantic active caching* techniques[1] and novel data structures and algorithms to quickly identify whether queries on integrated content can be answered completely or partially from cache. In the latter case, DICE can form the queries for the exact missing content.

We have developed the domain-independent mediator-based Content Integration System (Coins; http://netcins.ceid.upatras.gr/Software/DnC/), which includes the DICE cache subsystem, and implemented several applications on top of it. In this article, we describe DICE's overall architecture and present HyperHotel as an example application to show how our system can be used in the field. We also discuss performance results that show the system's efficiency.

 1089-7801/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

Related Work in Dynamic Content Caching

Researchers have extensively studied the problem of answering queries by using materialized views — the query-containment phase of the semantic-caching scenario — in the contexts of query optimization and data-integration and data-warehouse design. (The problem is known to be NP-complete.[1]) With our techniques, we solve the related problem of obtaining the maximal set of cached results for each query. Our algorithm is similar to, but less complex than, Pottinger and Levy's[2] because all queries in our setting are over the same set of relations and involve all the joined relations' attributes.

Our work is most closely related to that of Luo and Naughton.[3] However, DICE can produce remainder queries for the complete and exact missing results to a given query, while their work uses heuristics to provide approximate answers to user queries. Furthermore, whereas Luo and Naughton process only simple conjunctive queries, DICE can also handle queries with predicates connected by AND and OR and that might include a constant number of joins and comparison predicate values.

Our techniques and algorithms further guarantee that we always get the maximum out of our cache by combining data from multiple cached queries; the Luo and Naughton approach uses data from a single cached query each time. Using novel data structures and Bloom filters, DICE can identify hits and misses and produce the remainder (missing) queries to locate the full results. In contrast, Luo and Naughton use linked lists for lookups and file-system-based data for storage and retrieval.

References

1. A. Levy et al., "Answering Queries Using Views," *Proc. ACM Symp. Principles of Database Systems* (PODS), ACM Press, 1995, pp. 95–104.
2. R. Pottinger and A.Y. Levy, "A Scalable Algorithm for Answering Queries Using Views," *Proc. Int'l Conf. Very Large Databases* (VLDB), Morgan Kaufmann, 2000, pp. 484–495.
3. Q. Luo and J. Naughton, "Form-Based Proxy Caching for Database-Backed Web Sites," *Proc. Int'l Conf. Very Large Databases* (VLDB), Morgan Kaufmann, 2001, pp. 191–200.

DnC Architectural Overview

Coins uses off-the-shelf wrapper-generation toolkits and standard XML technologies to extract and integrate information from Web sites. It also incorporates an XML-based semantic metadata repository, created and maintained by the Coins' administrator, including declarative descriptions of the source sites' content and capabilities.

DICE and Coins — collectively referred to as DnC — target content-integration issues representative of those found in e-commerce applications that compare products, hotels, online auctions, and so on. Such systems are characterized by:

- the need to translate the back-end sites' data models into a single overarching data model;
- the fact that queries posed to the system refer to all relations in the unified data model and involve all attributes in the joined relations; and
- the fact that the unified model and query semantics are available a priori at the mediator.

Using a mediator-based system to deploy such applications avoids obsolete techniques built around data entry. However, we decided to introduce DICE between the mediator and end user in order to minimize network accesses, and thus, the network overhead.

As shown in Figure 1, DnC's multitier architecture includes

- front-end subsystem,
- a caching subsystem (DICE), and
- an XML-enabled mediator subsystem.

For maximum interoperability and code reusability, all DnC components communicate via XML documents constructed according to a pair of predefined DTDs: `Query` and `Response`. These DTDs define the interface between any two DnC components.

In processing a typical user query, DnC performs the following steps:

- The front-end modules accept user queries through an HTML-based interface and convert them to Query-DTD-compliant (QDC) XML form. These query-generator modules then forward the QDC-XML documents to the DICE caching subsystem for further processing.
- DICE checks its internal data structures to see if the query can be answered from cache. For a full hit, DICE returns the cached result to the front-end modules. For a partial or full miss, it constructs a set of queries representing the missing data in QDC-XML form and forwards the resulting document to the mediator subsystem.
- The mediator subsystem uses its semantic metadata repository to select the set of back-end sites to contact and retrieves the relevant Web pages. After the wrapping process and a preliminary query processing on the mediator's behalf (to eliminate duplication, integrate

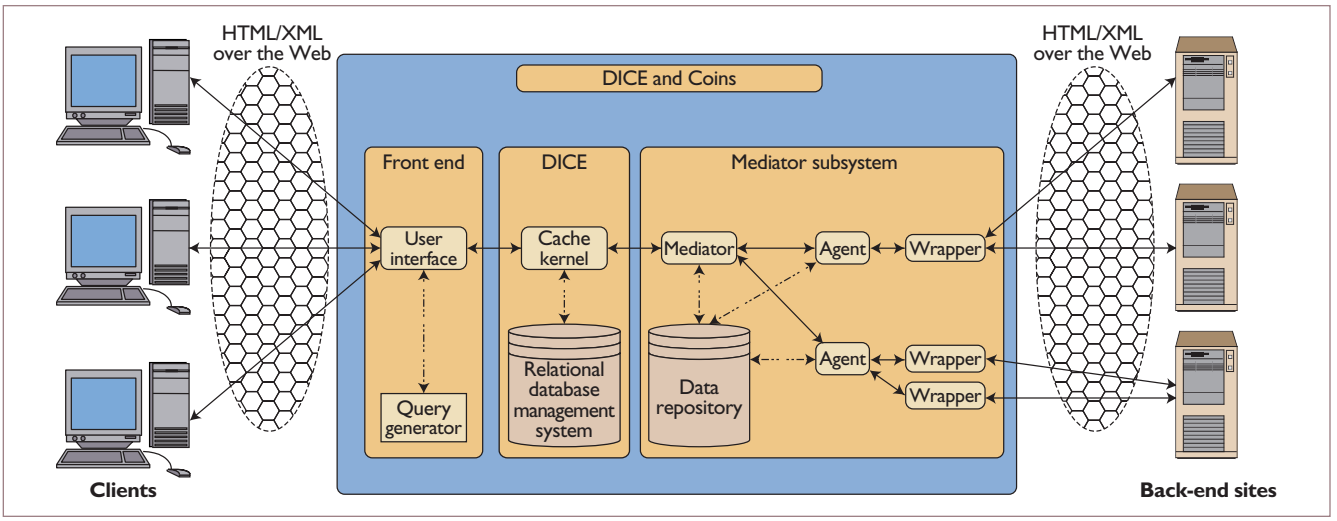

Figure 1. DICE-and-Coins architecture. Coins consists of three distinct but cooperating components: the front-end subsystem, the Data Integration Cache Engine (DICE), and a mediator.

semantically identical tuples to a common format, and so on), the mediator subsystem returns the resulting Response-DTD-compliant (RDC) XML documents to DICE.
- DICE stores the newly acquired data and updates its internal data structures. It then combines all the relevant cached tuples to construct a single RDC-XML document, which it returns to the front-end modules to format the output and create appropriate HTML pages in response to the initial user query.

Now that we have a broad overview of the DnC system, we can delve into DICE's design and implementation to understand the workings of this key subsystem.

Data Integration Cache Engine

Figure 2 (next page) shows an overview of the DICE architecture. The core idea is to use summarizations of cached entries to identify full and partial cache hits of query results and to compute the set of *missing queries* — those whose results represent the data items not present in the cache. DICE uses a relational database management system (RDBMS) to store cached information as tuples within relations. It then represents cached queries as *materialized views* over these relations. The cache summaries are created by using the materialized views, Bloom filters,[2] and specialized data structures in the hit/miss-detection algorithm and in the production of (sub)queries corresponding to the exact missing content, all of which greatly reduce the cost of performing these procedures.

Bloom filters provide a nice way to summarize distinct values. A Bloom filter comprises a bit vector and a set of hash functions that map input values to positions in the bit vector. Data that is to be added to a Bloom filter is hashed using these functions, and the corresponding bits in the bit vector are set to 1. We can thus check a value's existence in a Bloom filter by taking the same hashes and checking whether all the corresponding bits are indeed set to 1.

By using query rewriting and materialized views over cached entries, DICE can extract the maximally contained cached query results (that is, any and all cached data relevant to the current query) with minimal complexity. Our caching framework provides the mediator subsystem with the exact set of missing queries and delivers the exact query response to the front-end modules (full semantic active caching).

DICE can handle queries of the following form:

- `SELECT` projection_list `FROM` relations
- `WHERE (NOT)` atomic_constraint (attribute, value)
- `[AND|OR] (NOT)` atomic_constraints
- `AND (NOT)` comparison_constraint
- `[AND|OR] (NOT)` comparison_constraints

In the class of applications targeted by DnC, the `projection_list` includes all attributes contained in the union of the tuples in each of the relations (that is, all possible attributes).

We process atomic constraints (predicates containing only equality constraints) separately from predicates with comparison (≤ or ≥) constraints. This

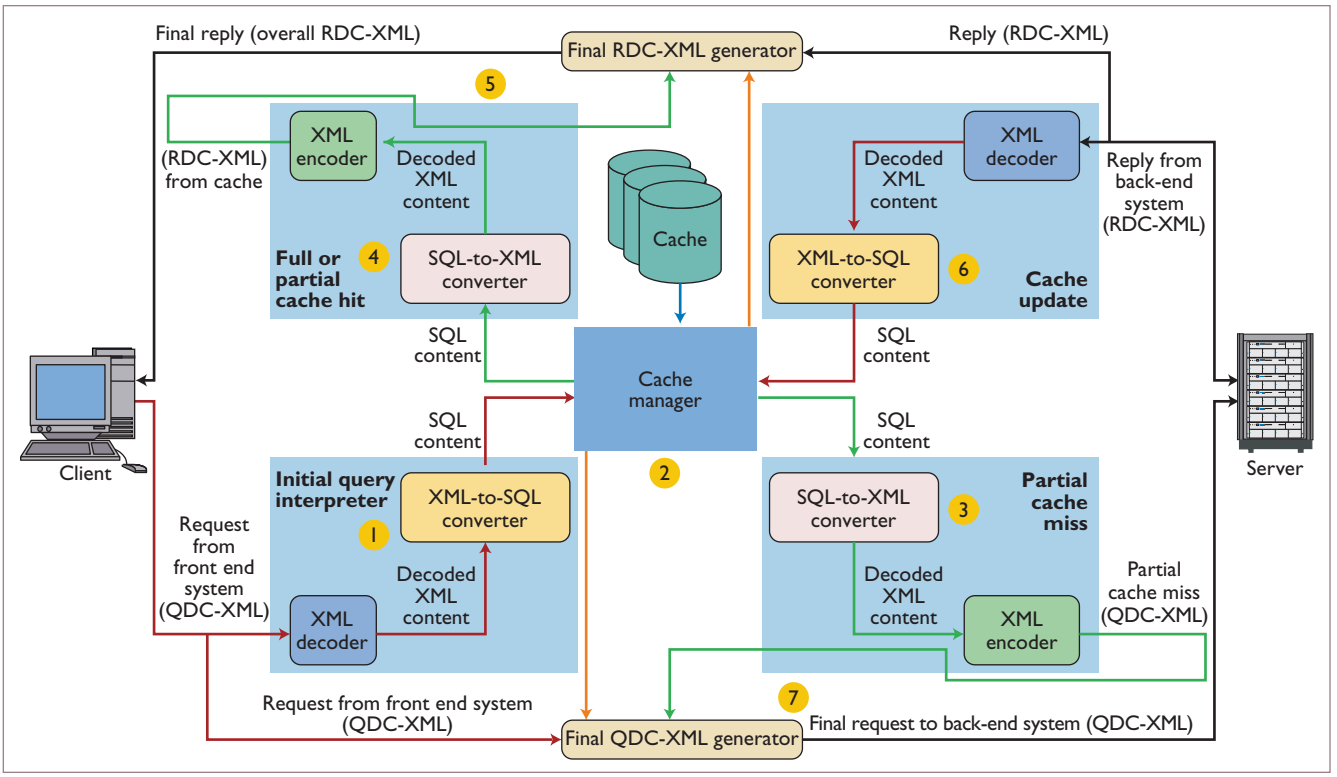

Figure 2. DICE cache manager architecture. As a user query moves into the cache subsystem, DICE: (1) translates it into an internal form; (2) checks for partial and full cache hits; (3) creates queries for missing content as necessary; (4) gathers cached data; (5) constructs a preliminary response document, and (6) incorporates data integrated by the mediator. If the query was a full miss, DICE (7) forwards it to the mediator and moves on to steps (5) and (6). Note that the use of XML and the `Query` *and* `Response` *DTDs lets us reuse components such as the XML encoders and decoders and the converters for moving between SQL and XML.*

is because Bloom filters can effectively summarize distinct values (equality constraints), but ranges of values (comparison constraints) require additional mechanisms. We thus separate query predicate constraints into two major categories:

- *Atomic-attribute-value predicate combinations of constraints* (AACs) — the set of all atomic constraints in the query expression — take the form `attribute=value`.
- *Comparison predicate constraints* (CPCs) always take the form `attribute {&|≥} value`.

As mentioned, a Bloom filter consists of a bit vector and a set of hash functions. To minimize the possibility of more than two input values hashing to the same bits in the filter's vector, we use a cryptographic hash function (namely, MD5): to insert or look up an entry in a Bloom filter, we first hash it to create a 128-bit pseudo-unique number, which we divide into equal segments for DICE to use as indexes in the filter's bit vector.

In a new filter, all bits are set to 0 (indicating that the filter contains no items). For an *insertion* operation, DICE sets the corresponding bits to 1 in the filter's bit vector. (Note that a single bit could be set to 1 more than once because some of the filter's hash functions might be identical for two values. However, we minimize that possibility by using MD5 and tuning it by varying the bit vector). For a lookup operation, the response is the bitwise `AND` of all relevant bits.

Query Bloom filter. DICE represents the set of cached queries using a *query Bloom filter* (QBF). DICE uses semantic information (supplied at the configuration step, as described earlier) to apply a predefined ordering to the positions of projection attributes, relations, and query predicates in incoming queries. (Remember that in the class of applications our system addresses, queries are over all relations and attributes, and that query semantics are known a priori; thus, a simple lexicographic ordering of the relations and attributes is suffi-

HyperHotel: An Example Application

To showcase the DICE and Coins (DnC) architecture, we developed HyperHotel (http://netcins.ceid.upatras.gr/Software/DnC/HyperHotel/). This data-integration application brings together vast amounts of unstructured and semistructured information available online from real-world hotels (location information, room rates, availability, online booking, and so on). Our main goal was to create a semiautomated way to uniformly query all such facilities. We aimed to surpass currently available paradigms based on data entry and proprietary communication protocols between service providers and back-end sites.

HyperHotel lets users search across multiple sites. User queries include such attributes as hotel location, hotel class and facilities, number of beds per room, desired stay length, and desired cost range. Constraints concerning stay and desired cost are range-based, while all other constraints are equality based (that is, constraints on dates are of the form $date_i$ $\{\leq|\geq\}$ $value_i$, while constraints on hotel class are of the form $hotel\_class_i = value_i$).

This type of application is a perfect match for DnC. Content integration provides end users a uniform mechanism for selectively querying multiple hotel Web sites, and DnC's interface makes it easy to compare between candidates. Coupled with DICE's advanced features, the fact that DnC fetches and integrates content on demand guarantees the timeliness and correctness of returned results, while keeping query-completion time low.

cient.) It then uses the text representation of the resulting queries to perform a simple lookup in the QBF to check whether the incoming query is identical to one in the cache (a *full cache hit* in passive-caching terminology). In case of a *cache miss*, DICE decomposes the query into its AACs and CPCs.

AACs and Bloom filters. DICE stores the AACs for all cached queries in a second Bloom filter, called the *atomic attribute predicate Bloom filter* (ABF). The cache engine also maintains a hash table of all views, using the AACs as the keys in determining which cached materialized views answer which atomic attribute-value subqueries. DICE uses this scheme in cache-miss scenarios to locate the maximally contained results from the cache. It then creates the minimal set of queries whose union corresponds to the exact missing information, and delivers them to the mediator for processing.

There is a small chance of false positives when using Bloom filters to identify hits and misses. To avoid such errors, the kernel maintains a counter (instead of the one-bit identifiers) for each position in the filter's bit vector. DICE increments the counter each time it hashes a query to a location in the filter. If the counter values at all locations corresponding to an incoming query are less than one, the kernel assumes a false positive and forwards the original query to the back-end mediator subsystem. This simply results in the generation of an extra message to — and the retrieval of an extra object from — the mediator.

CPCs and paths. As mentioned, CPCs are predicate constraints using comparison ($\leq$ or $\geq$) operators. Such constraints are usually found in queries for ranges of values (for example, "find all hotels with a per-night price between $100 and $200"). Bloom filters are not appropriate for storing ranges of values; each value stored in a Bloom filter must be constant, distinct, and randomly distributed in the filter's bit vector. Hence, the cache engine also maintains data structures as a tree of *paths* — lists of ordered value pairs that correspond to the lower and upper bounds of comparison predicates. This tree of paths provides the cache engine with the range of values, if any, in each cached materialized view.

For example, CPCs might appear in the `Dates` and `Prices` query fields in the application described in the sidebar, "HyperHotel: An Example Application," because users are generally interested in vaguely defined time periods and price ranges. For each AAC in ABF, the kernel maintains an ordered list of `Date` nodes. For each path in the date list, there is a corresponding ordered list of `Price` nodes. Thus, each CPC is represented as a list of value-pair lists. DICE splits or merges the paths so that no overlapping occurs between them.

Deployment Issues

DICE requires a priori semantic knowledge of the queries it receives from the system's users. Furthermore, to enable active caching, the cache administrator must provide DICE with all possible values for each atomic attribute in the query expression, as well as the minimum and maximum values for the attributes that contain comparison predicate values. Manual configuration of this kind is very similar to that required in the field of proxy servers; when configuring a regular proxy server, the administrator specifies which URLs to cache by adding them to a configuration file. Form-based

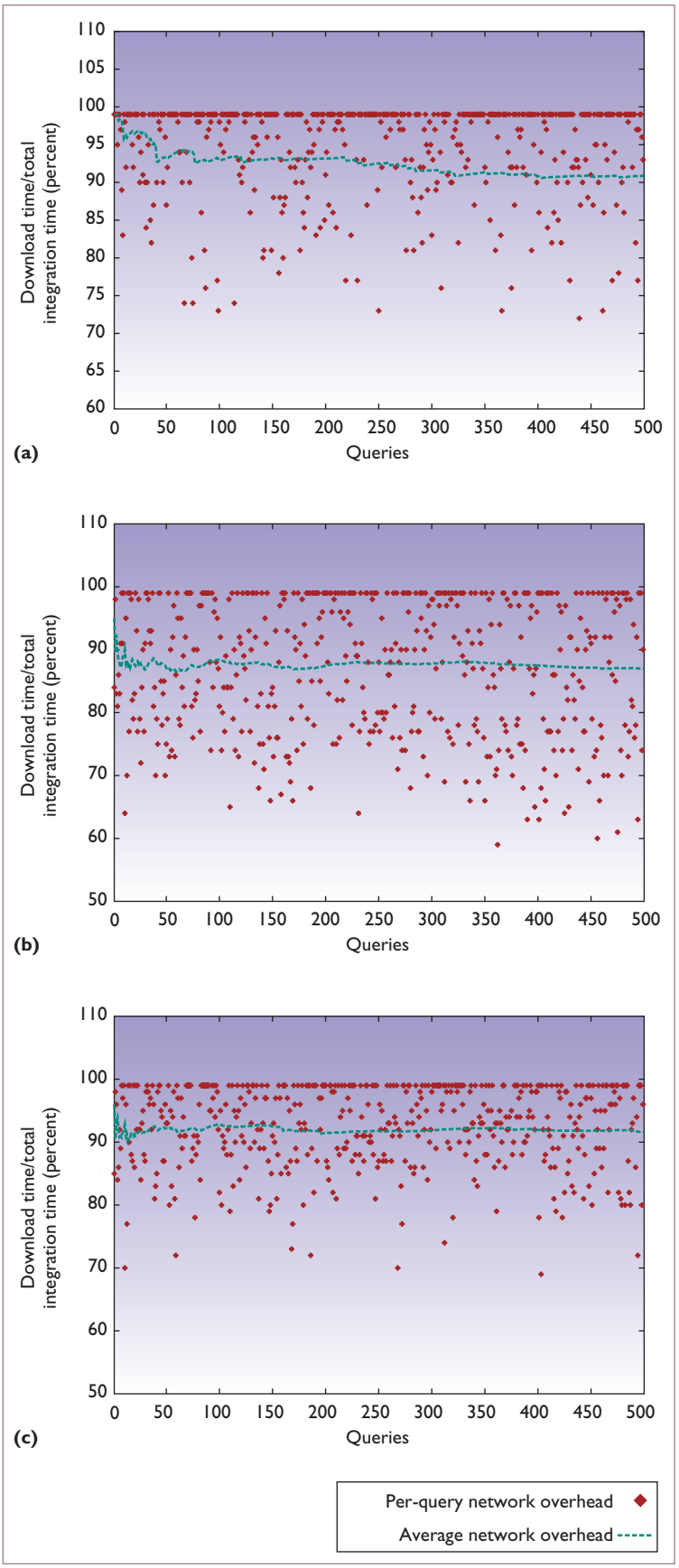

Figure 3. Network overhead expressed as a percentage of total data-integration time during data download. The charts show results for Coins in three usage cases: (a) With the Data Integration Cache Engine (DICE) in place, (b) with the Squid cache engine in place for static caching, and (c) with no cache engine in place. Distinct points correspond to queries, and the green line shows the average overhead.

proxy caching systems also require a priori specification of the cacheable forms.[3,4]

All queries that DICE handles refer to the same set of data sources, which greatly simplifies the process of answering queries using the cached views. Unlike some related applications,[5–7] the class of applications supported by our data-integration infrastructure requires no complex, generalized algorithms to solve the problem of query containment (known to be NP-complete).[5] Because a fixed number of relations participates in each query (that is, a constant number of joins), we do not try to rewrite the query as a conjunction of the relevant views. Instead, we use the ABF and the functionality provided by the relevant data structures to work from the maximal result set in each relevant view.

Performance Study

Although our approach's novelty is in the DICE caching subsystem, we choose to analyze its performance in conjunction with the mediator subsystem — specifically, testing DnC's performance operating within a real-world application such as HyperHotel. This will also stress our solution's appropriateness for the domain of content-integration applications.

Benchmarking

We believe mediated data represents a harder case for caching mechanisms than static information or simple SQL queries over a local database, because the queries' semantics cause variations in the set of sites to contact on every user request. As a direct consequence of this complexity, traditional workloads — such as the TPC-W benchmark (www.tcp.org/tcpw) — aren't applicable for benchmarking DnC because they focus primarily on performance aspects of the overall system rather than the cache in particular. We thus created a synthetic workload to measure all of the system's query-execution stages:

- Front-end modules' interactions with the

caching subsystem (via Java RMI);
- Hit/miss detection and remainder-query generation (if required);
- Interactions between the caching and integration subsystems (via Java RMI);
- Information retrieval and integration (mediation); and
- Cache update and query-result generation.

We implemented a specialized query generator to simulate real-world user behavior, which is by no means uniform with respect to HyperHotel's various query attributes (four-star hotels are more popular than two-star facilities, for example). We associate a probability with each possible attribute value, which the query generator uses to randomly choose values.

In the content-integration stage of query execution, network-intensive information retrieval forms the main bottleneck. Remember that generating remainder queries can increase the number of queries posed to back-end sites, and that the amount of data downloaded depends primarily on the queries' quantity, not quality. Therefore, a workload that causes many partial misses or hits presents greater difficulty than a workload that generates full cache misses or hits, because of the network overhead it inflicts.

We executed all benchmark tests on a 733-MHz Pentium III PC, with 256 Mbytes of memory and a 2-Mbps network connection, running Debian Linux 3.0 and using the Java Developer Kit versions 1.3.1 and 1.4.1. We performed several workload runs of 100, 200, 500, 1,000, and 1,500 queries each. Due to space considerations, we present results only for the 500-query test case, but we obtained very similar figures for the other test cases.

To provide reference points, we also ran the tests

- using an off-the-self Web proxy cache engine, and
- without any caching.

For the former, we used the Squid proxy cache (www.squid-cache.org), which we placed between DnC and the Internet so that all requests over the Web passed through it. (For performance metrics comparing Squid to other caching products, see http://cacheoff.ircache.net.) We explicitly provided Squid with enough storage space to ensure that it wouldn't invalidate cached data during our benchmark runs.

Because the mediation tasks are network-bound, we opted to benchmark network traffic, network overhead (that is, the ratio of data-retrieval time to total content-integration time, which includes the time spent at the wrappers, mediator, and network), and back-end server access. To focus on the caching subsystem's performance, we also measured each query's execution time.

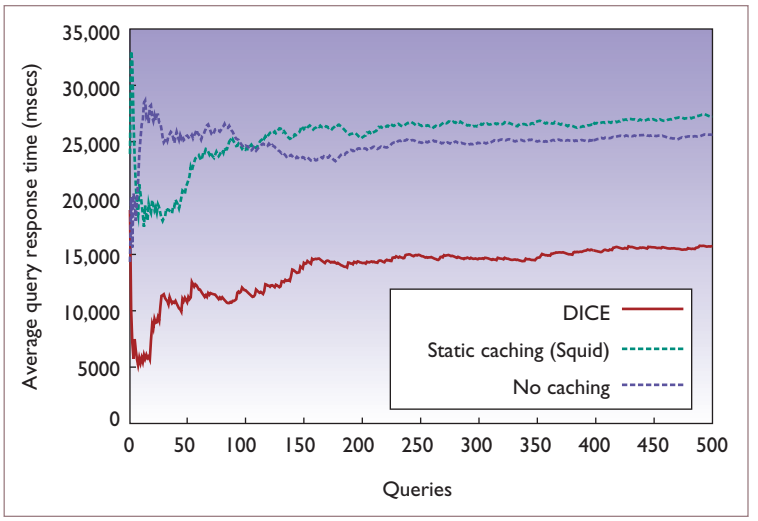

Figure 4. Average query-response times. DICE cuts the average turnaround time for queries to almost half of that needed without caching, whereas Squid, which is optimized for static data, adds overhead.

Results

Our analysis of the benchmark results reveals several facts. As Figure 3 illustrates, data-retrieval constitutes 90 percent of the total data-integration time (by far the most time-consuming phase of query processing) in all three approaches — active caching with DICE (Figure 3a), static caching with Squid (Figure 3b), and no caching (Figure 3c). The other 10 percent includes the time for wrapping, integration, and preliminary reply processing on the mediator's behalf, which proves that network overhead is the main bottleneck.

With DICE, queries took an average of 38.7 percent less time to complete than without caching and 42.3 percent less than the static caching scheme (see Figure 4). Indeed, because data is drawn from back-end Web sites using their HTML form-based interfaces, the static caching scheme merely adds extra overhead to query processing.

As Figure 5 (next page) shows, our caching scheme averaged just half the back-end network accesses that the other caching schemes required. Thus, DnC creates a much lower back-end server load.

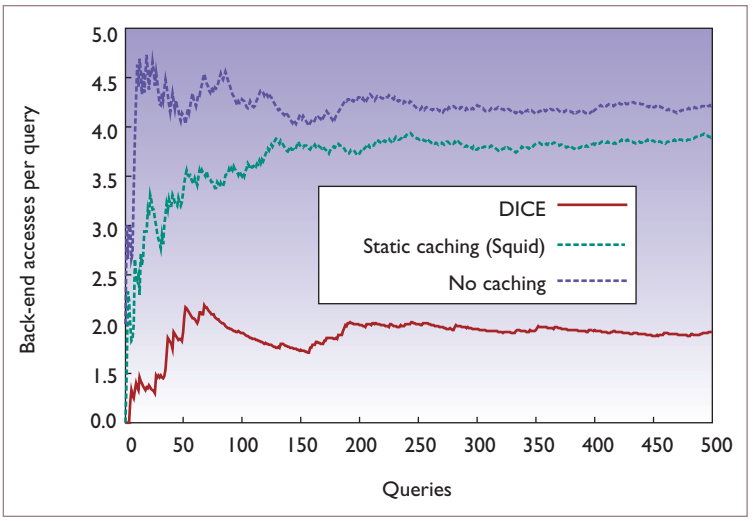

Figure 5. Average back-end accesses per query. DICE cuts the average number of back-end accesses almost in half compared to results with no caching; Squid reduces accesses only for "identical" queries — that is, only for queries with the same values in all (atomic and comparison) constraints.

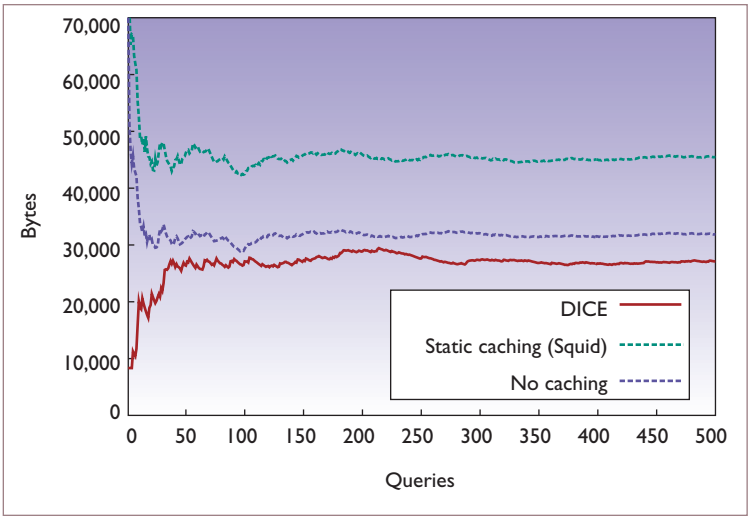

Figure 6. Average bandwidth per query. DICE decreases the average number of bytes transferred per query to 85 percent of the volume transferred when no caching is in place. The results for Squid (over which DnC showed a 40 percent improvement) are misleading due to the Squid engine's use of session-specific information, which leads to a somewhat artificial increase in the raw byte count.

As Figure 6 demonstrates, our caching scheme causes an average of 15 percent less network traffic than with no caching, and 40 percent less than with the static caching scheme. However, Squid rewrites cached pages so that they contain session-specific information (such as session IDs), which is usually kept in cookies at the browser. This leads to a somewhat artificial increase in the raw byte count of data fetched over the Web because we can measure only the results Squid returns, and thus must use the "augmented" Web pages.

Conclusions

Cache consistency is a key issue when working with data gathered from Web sources. However, it is highly dependent on the application semantics; for example, a scheme that would do well under a Web-caching scenario might fall apart if applied to a database environment. We believe that some form of invalidation mechanism (such as time-to-live, as in Web-proxy caching) will fit well with the rest of our system. Push–pull[8] techniques or approaches based on expiration mechanisms[9] also represent opportunities for future extensions to our framework.

We would also like to experiment with distributing our system's caching and query-processing chores across multiple computers on the Internet. Our use of an RDBMS at the storage layer simplifies migration to a distributed environment, but it leaves many open issues: optimal forwarding and redirecting of queries between multiple DnC servers, optimal data placement, and DICE state consistency are just a few of problems that would arise in a distributed world. We believe that borrowing techniques from the field of content distribution networks and distributed databases might help us in this direction.

References

1. P. Cao, J. Zhang, and K. Beach, "Active Cache: Caching Dynamic Contents on the Web," *Distributed Systems Eng.*, vol. 6, no. 1, 1999, pp. 43–50.
2. B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, 1970, pp. 422–426.
3. Q. Luo et al., "Active Query Caching for Database Web Servers," *Proc. Int'l Workshop on the Web and Databases* (WebDB), Springer-Verlag, 2000, pp. 29–34.
4. Q. Luo and J. Naughton, "Form-Based Proxy Caching for Database-backed Web Sites," *Proc. Int'l Conf. Very Large Databases* (VLDB), Morgan Kaufmann, 2001, pp. 191–200.
5. A. Levy et al., "Answering Queries Using Views," *Proc. ACM Symp. Principles of Database Systems* (PODS), ACM Press, 1995, pp. 95–104.
6. S. Chaudhuri et al., "Optimizing Queries with Materialized Views," *Proc. Int'l Conf. Data Eng.* (ICDE), 1995, IEEE CS Press, pp. 190–200.
7. S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting Aggregate Queries Using Views," *Proc. ACM Symp. Principles of Database Systems* (PODS), ACM Press, 1999.
8. P. Cao and C. Liu, "Maintaining Strong Cache Consistency

in the World Wide Web," *IEEE Trans. Computers*, vol. 47, no. 4, 1998, pp. 445–457.

9. A. Dingle and T. Partl, "Web Cache Coherence," *Computer Networks and ISDN Systems*, vol. 28, no. 7-11, 1996, pp. 907–920.

Peter Triantafillou is a full professor at the University of Patras, Greece. His research interests include large-scale systems for Internet content delivery, sharing, and integration, with particular emphasis on peer-to-peer systems and distributed, event-based publish–subscribe systems. He received a PhD in computer science from the University of Waterloo, Canada. Triantafillou is the director of the Network-Centric Information Systems (NetCINS) lab at the University of Patras, and a senior researcher with the Research Academic Computer Technology Institute, Greece, where he heads the research group on Network-Centric Information Systems. Contact him at peter@ceid.upatras.gr.

Nikos Ntarmos is a PhD student at the University of Patras, Greece. His research interests include content integration and dissemination over the Web, active and semantic query result caching, and peer-to-peer systems, with emphasis on issues of accountability, routing, security, load balancing, and integration with ubiquitous computing. He received an MSc in computer science from the University of Patras, and a Diploma in computer engineering from the Technical University of Creete, Greece. Ntarmos is a researcher with the NetCINS lab at the University of Patras, and with the Research Academic Technology Institute, Greece. Contact him at ntarmos@ceid.upatras.gr.

John Yannakopoulos is an MSc student at the University of Crete. His research interests include miniaturization of networked computers and embedded systems, novel runtime system support, and communication protocols for miniature, real-time, and distributed computer systems. He received a Diploma in computer engineering from the Technical University of Creete, Greece. He currently is a research assistant in the Institute of Computer Science at the Foundation of Research and Technology, Hellas (ICS-Forth), Greece. Contact him at giannak@ics.forth.gr.