D.I.C.E. and Co.In.S. : A Data Integration Cache Engine for a Content Integration System

Peter Triantafillou and Nikolaos Ntarmos
Comp. Eng. & Informatics Dept.
University of Patras
<peter,ntarmos>@ceid.upatras.gr

John Yannakopoulos
Computer Science Dept.
University of Crete
*<giannak@ics.forth.gr>

Abstract

Content integration of web data sources is becoming increasingly important for the formation of the next generation information systems. A common performance bottleneck faced by all proposed solutions is the network overhead incurred when contacting the integrated e-sites. With this paper we contribute ongoing work on D.I.C.E. and Co.In.S.; a domain-independent Content Integration System and its Data Integration Cache Engine. DICE constitutes a cache engine utilizing novel techniques for operating as a fully active semantic cache. We show how our contributions can be applied in the field of content integration in order to improve the response time of content integration systems, representative of a large class of e-commerce applications. We have implemented the proposed architecture and are currently developing a number of applications.

1 Introduction

Web data integration has become vital for modern data management systems. Most relevant research focuses on the wrapping part of the integration process. In the real-world, however, the bottleneck of the system is the network overhead, incurred when contacting the integrated e-sites.

Co.In.S. - the Content Integration System, is a domain-independent mediator-based system. With this work we contribute D.I.C.E.; the Data Integration Cache Engine, operating in the context of Co.In.S. DnC[1] targets content integration systems, representative of those needed to support a large class of e-commerce applications (e.g. comparative e-shopping for books or CDs, e-hotels, online auctions, etc.), with such characteristics as: (i) a need for translation of the large variety of different data models, used by the back-end e-sites, into a single data model, (ii) the fact that queries posed to the system are over all relations of the unified data model and involves all attributes in the joined relations, and (iii) the fact that the unified model and query semantics are available a-priori at the mediator.

[1]We use DnC to refer to D.I.C.E. and Co.In.S., where appropriate.

2 D.I.C.E. and Co.In.S

2.1 Architectural Overview

The architectural emphasis is on speed, platform independence, and ease of deployment. Briefly, DnC is a multi-tier architecture, consisting of the following components: (i) the user interface, (ii) the Front-end modules, implemented using Java Servlets and XML-related technologies, (iii) the caching subsystem (aka D.I.C.E.), based on off-the-self database management systems and novel cache management techniques, (iv) an XML-enabled mediator subsystem, utilizing third-party wrappers and standard XML-derived technologies, and (v) the back-end e-sites. All internal communications between the components of DnC - that is, queries and their responses - use XML documents, constructed according to a set of *Query* and *Response DTDs* respectively[2].

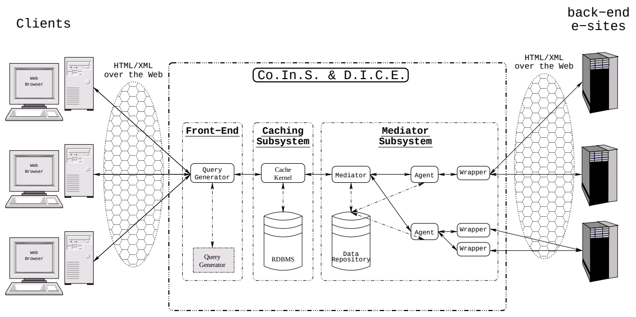

Figure 1. DnC Architecture

[2]We'll use the terms *QDC-XML* document and *RDC-XML* document to refer to a *Query/Response DTD Compliant* XML document respectively.

2.2 D.I.C.E.: A Macroscopic View

An effective and transparent method for tackling the network overhead is to cache queries and their results within the data integration system. D.I.C.E. performs this task in our setting. It uses a RDBMS at its storage layer. Cached information is stored in tuples in the RDBMS's relation and cached queries are represented in the cache as materialized views over the database relations.

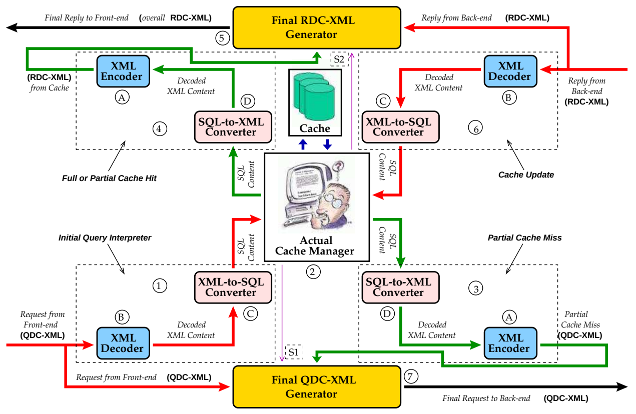

Figure 2. D.I.C.E.: The Cache Manager

A core idea of D.I.C.E. is the use of summarization of cached entries, based on a novel utilization of Bloom filters, along with specialized data structures, and of materialized views, in order to swiftly identify full and partial cache hits of query results, and to compute the set of subqueries whose results represent the data items not present in the cache (called "missing queries"). Our approach allows for very fast hit/miss identification and missing query generation. We'd like to stress that our caching framework provides the mediator with the **exact** set of missing queries and delivers the **exact** query response to the front-end modules (full semantic active caching).

The characteristics of queries that can be handled by our cache are shown in fig. 3. We assume that the projection_list includes all attributes contained in the union of the tuples in each of the target_relations (i.e. all possible attributes).

SELECT *projection_list*
FROM *target_relations*
WHERE (NOT) *atomic_constraint (atomic_attribute, value)*
[AND | OR] (NOT) *other_atomic_constraints*
AND (NOT) *comparison_predicate_constraint*
[AND | OR] (NOT) *other_comparison_predicate_constraints*

Figure 3. Class of queries supported by D.I.C.E.

Cache consistency and temporal coherency are issues typically discussed in works focusing on caching of data. We presently adopt some form of invalidation mechanism (e.g. TTL-based a la web proxy caching). An extended discussion of these issues is outside the scope of this paper.

2.3 Request Processing

1. The front-end module exports, an HTML-based query interface. The user contacts the server, through her favorite web browser and poses her query. The front-end modules convert the user request to a QDC-XML document describing the query, and forward it to the caching subsystem.
2. The caching subsystem checks its data structures for a cache hit. In the case of a full hit, the cached result is returned to the front-end modules. In the case of partial/full miss, the caching subsystem constructs a set of subqueries, representing the missing data, and forwards them to the mediator.
3. For each such subquery, the mediator selects the set of back-end sites that have to be contacted, and retrieves the relevant web pages. After the wrapping process, the translation of information to the system's internal data model, and a preliminary processing on behalf of the mediator (e.g. duplicate elimination, format conversion, etc.), the resulting RDC-XML documents are returned to the caching subsystem.
4. The caching subsystem stores the new data and updates its internal structures. It then constructs a single RDC-XML document, combining all of the relevant cached tuples, and returns it to the front-end.

The front-end modules pretty-format the output and create appropriate HTML pages with the answers.

3 D.I.C.E.: A Microscopic View

3.1 Bloom Filters

To insert or lookup an entry in a bloom filter, we hash the entry using the MD5 message digest generation algorithm, generating a 128-bits number. We then divide those bits into $k = 8$ equal segments of $n = 16$ bits long (i.e. into 8 short integers). Each of the k integers, is used as an index in the filter's bit vector. Hence, the length of the filter's bit vector is $2^n = 65536$ bits. If the operation is a lookup, then the response is the bitwise "AND" of all relevant bits. If the operation is an insertion, then these k bits are set to '1' in the filter's bit vector. Note that a single bit in the filter's bit vector might be set to '1' more than once.

3.1.1 The Query Bloom Filter

The set of cached queries is represented in our system through a Bloom filter, called the **Query Bloom Filter** (or

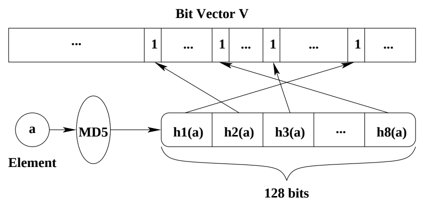

Figure 4. Bloom Filter

QBF). Assume that projection attributes, relations, and query predicates appear in the same order in every query; then the text representation of every query introduced to the cache is inserted in the QBF. Thus, we can immediately tell whether a query is *identical* to a stored one or not (a ”full cache hit” in the passive-caching terminology) with a simple lookup in the QBF. In case of a ”cache miss”, the query predicate constraints are decomposed into two major categories: atomic attribute - value predicate combinations of constraints (i.e. AACs) and comparison predicate constraints (i.e. CPCs); the former are of the form “$attribute = value$” (e.g. year=2003), while the latter are of the form ”$attribute \ \{< \mid >\} \ value$” (e.g. year>2003).

3.1.2 AACs and Bloom Filters

The AACs of all cached queries are similarly stored in a second Bloom filter, called the **Atomic Attribute Predicate** Bloom filter (or ABF). Additionally, the cache engine maintains a hash table of all views with AACs as its key, by which it knows which cached materialized views answer to which atomic attribute-value subqueries. By this scheme, in a cache miss scenario, D.I.C.E. is capable of obtaining the maximally-contained results from its cache, which is the best answer possible, and to extract the missing subqueries to be delivered to the mediator, in order to guarantee the completeness of the answer to the initial user's query.

Assume an incoming query Q. The DICE kernel transforms Q into Q', with all fields of Q appropriately sorted. The kernel then uses Q' and QBF to tell whether Q is identical to a cached query.

To avoid false positives, the kernel maintains a counter instead of only a bit for each location of the filter, incremented each time a query is “hashed” to a location in the filter. If the counters at all locations corresponding to an incoming query are > 1, the kernel assumes we have a false positive and simply forwards the original Front-end query to the Mediator subsystem for further processing.

Example 1 *(Query Bloom Filter). Assume the queries:*

- *Q_1 = select * from A, C, B where A.a = B.a and C.b = B.b and C.c = “$string_2$” and B.c = “$string_1$”, and*
- *Q_2 = select * from B, C, A where B.a = A.a and C.b = B.b and B.c = “$string_1$” and C.c = “$string_2$”.*

The queries are obviously identical. To use QBF, both $Q_{1,2}$ would initially be transformed into:

- *$Q'_{1,2}$ = select * from A, B, C where A.a = B.a and B.b = C.b and B.c = “$string_2$” and C.c = “$string_1$”.*

Then $Q'_{1,2}$ would be “hashed” to get the corresponding locations in QBF. If at least one of the counters is 0 then $Q'_{1,2}$ is not stored in the cache (i.e. full/partial miss). Otherwise, if all counters are > 1, the kernel assumes this is a false positive and forwards $Q'_{1,2}$ to the mediator subsystem. Finally, if all counters are > 0 and at least one of them is $= 1$, then there is a Q'', identical to $Q'_{1,2}$, stored in the cache (i.e. full hit).

If *none* of these counters is > 0, then the query is decomposed into its AACs and CPCs and further processed by the *Partial Cache Hit Service* module. This kernel module analyzes the query through the involved attributes and their constraints. Via the ABF[3], the kernel is capable of producing the equivalent or maximally contained rewriting of the requested query (answering query using the stored views), thus, to generate all or the maximum part of the response to it respectively. In the case of this second filter, false positive cases are extremely unlikely, although the analysis is outside the scope of this short paper.

3.2 CPCs and Paths

The situation is a little different in the case of CPCs. Note that ranges of values are ***not*** stored in the ABF. Bloom filters are not appropriate for storing range-based predicates; since each value stored in a Bloom filter is constant, distinct, and randomly distributed in the filter's bit vector. Hence, the cache engine also maintains data structures by which it knows the range of values provided by each cached materialized view. Thus, for each combination of (non-)equality constraints in the user queries, DICE maintains a list of lists of ordered pairs of values (i.e. *paths*), representing the ranges produced by the CPCs.

Assume, for example, that the mediated schema consists (i) of attributes where only equality or non-equality constraints are applicable (i.e. AACs), and (ii) of date/price information, where constraints are usually in the form of a range of values (i.e. CPCs). For each combination of AACs, we keep a list of lists of paths, as shown in fig. 5. For example date range $D1$ has associated with it price ranges/paths $(P1_s, P1_e), (P2_s, P2_e)$. Paths in these lists are split or merged appropriately, to avoid overlapping.

[3]Think of ABF as a “*bitmap representation*” of AACs of the stored materialized views. Remember that each value that is to be stored in a Bloom filter must be constant, distinct, and randomly distributed in the bit vector of the latter. We thus use separate data structures for storing and manipulating CPCs.

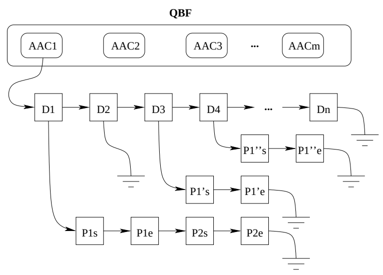

Figure 5. CPCs and Paths

3.3 Deployment Issues

The user poses a query in terms of the mediated schema that DnC exports initially through its front-end modules. The query is delivered to the $D.I.C.E.$ caching framework, where it's translated and formulated appropriately, in order to match with the cache's internal data sources. Thus, D.I.C.E. must know a-priori the semantics of the queries that are to be delivered to it.

Furthermore, to enable active caching, the administrator is required to provide $D.I.C.E.$ with the possible values for each atomic attribute that participates in the query expression. Additionally, for the attributes that contain comparison predicate values, the minimum and maximum value of the respective range must be provided. We need the minimum and maximum values of the range, because we use list semantics, i.e. the path which is defined by these two values is an *ordered list*. These values are provided in the configuration step.

We stress that such manual interventions are similarly required in proxy servers; its administrator specifies which URLs the proxy should cache, thru a configuration file. Moreover, form-based proxy caching systems ([3, 2]), also require an a-priori specification of the cacheable forms.

4 Related Work

The problem of answering queries using materialized views, that is the query containment phase of the semantic caching scenario, has been studied extensively, either in the query optimization or in the data integration and data warehouse design contexts and is known to be NP-complete [1]. With our techniques, we solve a related problem of obtaining the maximal set of cached results for each query, using an algorithm similar to [4] but less complex, since in our setting all queries are over the same set of relations and involve all attributes of the joined relations.

The work that is most closely related to ours, is that of Luo et al. [2]. We differ, however, in: (i) D.I.C.E. can produce remainder queries for the **complete** and **exact** missing results to a given query; (ii) D.I.C.E. can handle queries with predicates connected by both AND and OR and that may include a constant number of both joins and comparison predicate values, where the queries processed in [2] and in similar works, are simple conjunctive queries with neither joins nor comparison predicates; (iii) our techniques and algorithms guarantee that we always get the maximum out of our cache, combining data from multiple cached queries, as opposed to data from a single cached query each time; (iv) we use novel data structures and Bloom filters for the hit/miss identification and the production of the remainder (missing) queries, as opposed to linked lists for lookup and filesystem-based (files / directories) data storage and retrieval; (v) D.I.C.E. is used in the context of a content integration system, a context we believe to be both more demanding than caching queries over a single database (or website).

5 Conclusions

In this paper we have presented Co.In.S. - the Content Integration System - and D.I.C.E. - its Data Integration Cache Engine. We tackle the problem of query result caching in the context of web content integration systems, which are representative of a large class of e-commerce applications, backed by a fully functional implementation. Our main contribution is in active caching: the development of algorithms and of appropriate data structures for the generation of the **exact** remainder queries in a partial miss scenario and the consistent cache update and for extracting the maximally contained cached data items.

We have implemented Co.In.S. and D.I.C.E. and are in the process of developing several e-commerce applications and testing their performance. Early performance results show that the use of DICE leads to an approximately 50% decrease in both the user-perceived turn-around time and the load of the back-end e-sites, and to an average 40% decrease in the network bandwidth usage, compared to a barebone mediator and a mediator featuring a static cache.

References

[1] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. ACM PODS '95*.

[2] Q. Luo and J. Naughton. Form-Based Proxy Caching for Database-backed Web Sites. In *Proc. VLDB '01*.

[3] Q. Luo, J. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active query caching for database web servers. In *Proc. WebDB '00*.

[4] R. Pottinger and A. Levy. A Scalable Algorithm for Answering Queries Using Views. In *Proc. VLDB '00*.