Semantic Caching of XML Databases

Vagelis Hristidis
Computer Science and Engineering Dept.
University of California, San Diego
vagelis@cs.ucsd.edu

Michalis Petropoulos
Computer Science and Engineering Dept.
University of California, San Diego
mpetropo@cs.ucsd.edu

Abstract

We present a novel framework for semantic caching of XML databases. The cached XML data are organized using a modification of the incomplete tree [ASV01], which has many desirable properties, as incremental maintenance, containment decidability and remainder queries generation in PTIME. The modification we propose alleviates the exponential blowup observed in [ASV01] by partitioning the domains of the XML schema nodes in domain ranges. We also provide an upper bound on the total size of the conditional tree type of the modified incomplete tree (MIT), which describes the data stored in the MIT.

XCacher operates on top of XML databases and intercepts the query requests from a web server. We show how the MIT is maintained and how queries are answered by sending a complete and non-redundant set of remainder queries to the XML database. Finally we present a replacement algorithm for the MIT.

1 Introduction

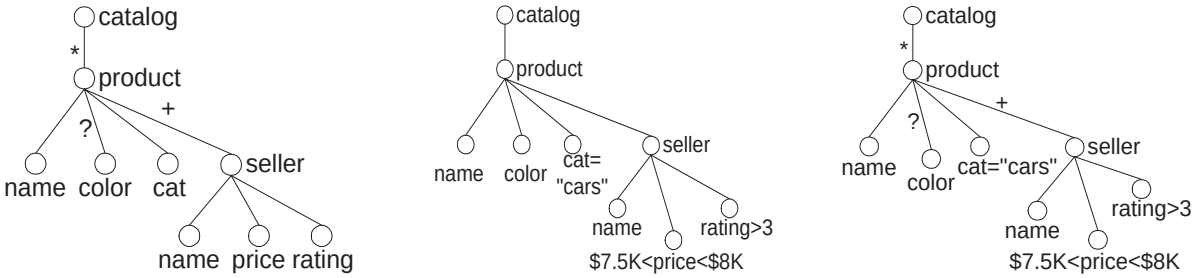
A considerable amount of work has focused on the problem of semantic caching for database-backed web applications [DFJ+96, LN01, GG99, LC99, Sel88]. These works focus on relational databases. As the number of web applications that are backed by XML databases increases, so does the need to provide efficient caching mechanisms that are suitable for the nature of the XML queries. However, the semantic caching approaches available, focus on specific classes of SQL queries [DFJ+96, LN01], which do not capture the navigational nature of the XML queries. XCacher is a system that facilitates semantic caching of XML databases for a subset of XQuery [W3C01], which is enough to support most reasonable applications.

In particular, XCacher operates on XML trees and supports XQuery queries that do not contain nested FOR-WHERE-RETURN expressions. Each query consists of an extract (FOR-WHERE clauses) and a construct part (RETURN clause). The extract part, which is cached by XCacher, is equivalent to a *prefix-selection query (ps-query)*, as defined in [ASV01], which selects a prefix of the source XML tree. The cache is organized as a *modified incomplete tree (MIT)*, which is based on the idea of the incomplete tree presented in [ASV01] and is an incomplete copy of the source XML tree. The representation of the cached data as an incomplete tree offers more flexibility and containment opportunities for subsequent queries than the traditional overlapping boxes [DFJ+96] employed so far to organize a semantic cache.

The MIT offers considerable advantages over the incomplete tree, i.e., more efficient maintenance and bounded size. Furthermore, the MIT retains the desirable properties described in [ASV01], i.e., incremental maintenance, containment decidability and remainder queries generation are in PTIME.

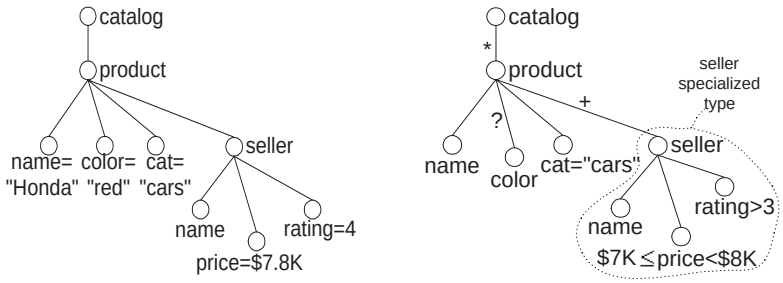
This work has the following contributions:

- A novel architecture is presented for performing semantic caching for applications backed by XML databases.
- A modification of the incomplete tree [ASV01] is presented and we show how the cache can be organized as a modified incomplete tree. The main drawback of the incomplete tree [ASV01] is the exponential blow-up of its size. We alleviate this problem by a modification that groups the range conditions of the queries into pre-specified *domain ranges* as described in Section 3.



(a) Schema tree (b) Query $Q1$ (c) τ_{Q1}

Figure 1: Schema and Q1



(a) T_d (b) $\tau \equiv \tau^N_{Q1}$

Figure 2: MIT after $Q1$

- An algorithm is described that creates in PTIME the remainder queries, which are guaranteed not to retrieve any data already in the cache.
- The MIT is stored in main memory, which bounds its size. A replacement algorithm is presented that removes the least recently used piece of data and its corresponding description from the MIT, when it gets full.

The paper is structured as follows. In Section 2 we present the framework and the architecture of XCacher. Section 3 describes the process of *refining* a MIT when a query arrives to the system. Section 4 explains how XCacher creates the remainder and the refinement queries for a query. An overview of the replacement algorithm is presented in Section 5. Finally in Section 6 we conclude and present future work directions.

2 Framework and Architecture

The architecture of XCacher is shown in Figure 3. XCacher operates on top of an XML database server that exports a view V. V is defined as a labeled tree [ASV01], whose structure is described with a schema tree S. The ordering of the children of V is unimportant and there is no distinction between attributes and subelements. Each node of V is assumed to have a unique *id*. The schema tree of Figure 1(a), that will be used as a running example, shows the schema of an auction database, where the catalog contains products that have a name, an optional color, a category they belong to, and a list of sellers. Each seller has a name, a price he/she is selling the product for, and a rating according to the customers' feedback.

XCacher runs as an application in the application server and a web server operates on top of the application server. Each web page p consists of a set of XQuery [W3C01] queries $q_1, \ldots, q_m$ that do not contain any nested FOR-WHERE-RETURN expressions. Each query q consists of an extract part q^E (FOR-WHERE clauses), which is served by the XML database, and a construct part q^C (RETURN clause), which is executed at the *query composer*. When q^C is executed against the result of q^E, it constructs a result equivalent to the result of the initial query q, that is, $q^C(q^E(V)) = q(V)$. q^E is a *prefix-selection query* (ps-query) [ASV01]. A ps-query browses the input tree V down to a certain depth starting from the root, by reading nodes with specified element names and possibly selection conditions on data values. The answer to a ps-query is a

prefix of V. For example the query of Figure 1(b) returns the products with category "cars", which have a seller, who sells the product between $\$7.5K$ and $\$8K$ and has a rating greater than 3.

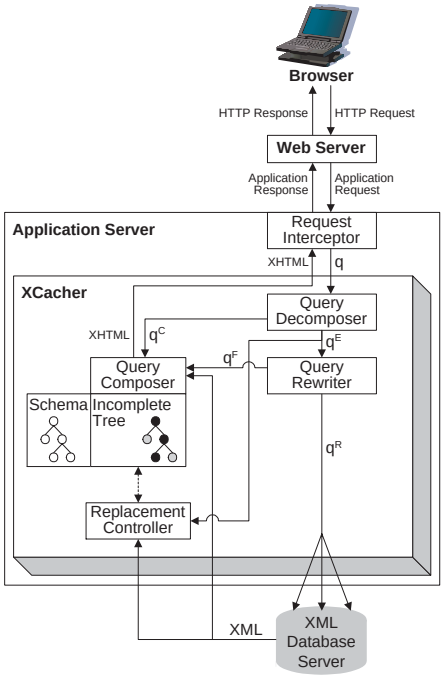


Figure 3: System Architecture

XCacher is responsible for the efficient execution of the extract queries q^E. The modules of XCacher shown in Figure 3 are the following: The *query decomposer* takes as input an XQuery expression q and outputs q^E and q^C. The *query rewriter* module takes as input the extract query q^E and determines if q^E can be answered entirely from the data stored in the cache, i.e., without accessing the XML database. If it can, the query rewriter outputs a refinement query q^F, which is the same as q^E and is executed on the data stored in cache. Otherwise, a refinement query q^F and a set S^R of remainder queries q^R are created and sent to the query composer and the XML database respectively, as described in Section 4.

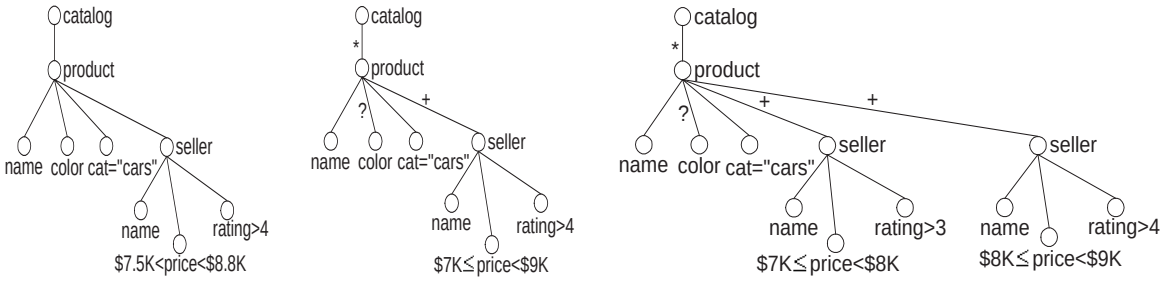
The answers to the remainder queries are input to the query composer and to the replacement controller, which is the only module that has a "write" privilege on the cache. If the new piece of XML data does not fit in the cache, the replacement controller decides, applying the replacement algorithm described in Section 5, which XML data fragment(s) to remove from the cache.

The query composer "merges" the results of the remainder queries and q^F, executes q^C on the result, and passes the output to the web server, as described in Section 4.

The cache is organized as a *modified incomplete tree (MIT)* $\mathbf{T}$, which consists of a *data tree* T_d which is a prefix of V and a *conditional tree type* τ which describes the information stored in T_d. In particular, a conditional tree type τ is a *specialization* of the schema tree S, i.e., a mapping of conditions to the schema nodes of S, where some schema nodes (which are called *clonable* below) are cloned to denote disjunction between their conditions. That is, the information stored in T_d for a node s of S is described by the union of all instances of s in τ. The conditions applied to the schema nodes in τ are equivalent to those of the ps-queries. Figure 2 shows (a) the data tree T_d and (b) the conditional tree type τ of the MIT after executing $Q1$. The generation of the conditional tree type is performed by the *replacement controller* module by applying the $RefineMIT$ algorithm presented in Section 3. The size of $\mathbf{T}$ is bound by the available main memory to provide fast access times.

The MIT differs from the incomplete tree defined in [ASV01] in the following ways: First, the conditional tree type of a MIT describes the information contained in T_d, in contrast to the incomplete tree, where the conditional tree type describes the missing information. This change was made because the em replacement controller, shown in Figure 3, needs the information of what is currently stored in the MIT to decide what to replace. Second, the *specialized types*[1] of an element s in τ, i.e., the descriptions of the data of type s contained in T_d, are disjoint. That is, there cannot be an element of T_d conforming to two different specialized types. This property simplifies the process of generating the remainder queries as it is shown below. Third, there is no mapping between the elements of the data tree of a MIT and the specialized types of the conditional tree type, because the specialized types are disjoint and the data tree is considered a "queryable" data source for the underlying query processor. In contrast, the incomplete tree generates a unique label $l(t)$ for each specialized type s' and all elements in T_d that conform to t have the same label $l(t)$. Hence the overhead imposed to maintain this mapping in a caching system with frequent replacements is avoided. Finally each specialized type t of τ is annotated with a *timestamps list* of the l last timestamps, when a query q, overlapping with t, arrived to the system. The timestamps lists are used by the replacement algorithm as described in Section 5.

[1] The specialized types are defined formally in Section 3



(a) Query $Q2$ (b) τ^N_{Q2} (c) Refined τ'

Figure 4: Refine process for $Q2$

The MIT is simpler and easier to maintain than the incomplete tree, but the desirable complexity results of the later still hold. In particular, the refine algorithm and the remainder queries' generation for a MIT $\mathbf{T}$ are more efficient because they do not access the data tree of $\mathbf{T}$, but only its conditional tree type. Furthermore the size of the MIT is bounded as it is shown below. Also, similarly to the incomplete tree, the following problems are in PTIME: incremental maintenance, containment decidability and remainder queries' generation. A drawback of the MIT is that it does not infer the emptiness of ps-queries with respect to the schema tree in order to avoid their execution. The performance overhead that is imposed though is minor since these queries are executed against the data tree of $\mathbf{T}$ that is kept in main memory and not against the XML database server that exports the view V.

3 Refine MIT

The $RefineMIT$ algorithm takes as input the schema tree S, the conditional tree type τ of the MIT $\mathbf{T}$ and a ps-query q and outputs a *refined* conditional tree type τ' that describes q in addition to the queries already described in τ.

$RefineMIT$ proceeds in three steps. The first step constructs the conditional tree type τ_q of q, by applying the conditions of q on the input schema tree S. Figure 1(c) shows τ_{Q1}. In the second step, the range conditions of τ_q are normalized according to the *domain partitions*, which are partitions of the active domains of the node values into *domain ranges*. For example, in τ_{Q1}, the range condition on the price has changed from $(\$7.5K, \$8K)$ to $[\$7K, \$8K)$, because the domain partition of price has domain ranges of $1K$ each.

The normalized conditional tree type τ^N_q of τ_q is defined as a conditional tree type, where each condition specifies a continuous set of domain ranges, which contains the range specified in q. According to this definition, the data tree represented by the initial conditional tree type τ_q is always a prefix of the data tree represented by the normalized conditional tree type τ^N_q. Figure 2(b) shows τ^N_{Q1}, which is the same as τ of $\mathbf{T}$, since $Q1$ is the first query that arrived to the system.

The third step of $RefineMIT$ computes the union of the input conditional tree type τ of $\mathbf{T}$ and the normalized conditional tree type τ^N_q of the query q, and outputs the refined conditional tree type τ':

$$\tau' = \tau \cup \tau^N_q \tag{1}$$

In order to compactly and efficiently merge the two conditional tree types, we must decide which nodes of τ will be further specialized, i.e., what parts of τ will be *cloned* to represent the overlap between the conditions of τ and τ^N_q. Specializations occur on *clonable* nodes, that are defined as a subset of the repeatable nodes of S. For example, in the schema tree of Figure 1(a), *product* and *seller* are clonable. A *specialized type* t is a specialization of a subgraph G of the schema tree S in τ, that is, an annotation of G with conditions. The root of G is a clonable node and the leaves of G are either clonable nodes or leaves of S. For example a *seller* specialized type is marked in Figure 2(b).

The basic idea of the algorithm that merges the two conditional tree types, which is not described in detail due to lack of space, is the following: We traverse τ and τ^N_q top-down until we find the first clonable node and if there is no specialized type t in τ that contains the corresponding specialized type t_q of τ^N_q, the current node of τ is cloned and annotated with the conditions of t_q. If t_q is contained in a specialized type

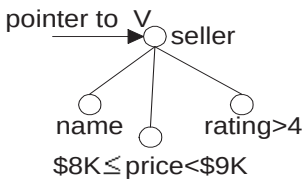


Figure 5: Local ps-query

t of τ, we continue with the first clonable descendant of t. If t_q overlaps with t then the current node of τ is cloned and annotated with the conditions of t_q such that the new specialized type does not overlap with t.

For example, Figure 4(a) shows a query $Q2$ submitted after the query $Q1$ of Figure 1(a). If τ and τ^N_{Q2} are the ones in Figures 2(b) and 4(b) respectively, then the $RefineMIT$ algorithm generates the τ' that is shown in Figure 4(c). Traversing τ^N_{Q2} and τ top-down, and comparing their specialization, no difference is observed until the *seller* specialized type. The *seller* specialized type t_{sel} of τ overlaps with the *seller* specialized type t^{Q2}_{sel} of τ^N_{Q2}, so *seller* is cloned and a new specialized type describing the difference of t_{sel} from t^{Q2}_{sel} is created in τ'. Note that the specialized types of τ' are non-overlapping.

Finally, the current timestamp is added to the timestamp lists of all specialized types that overlap with the current query q. When a specialized type t is cloned, then all the clones inherit the timestamps list of t.

If we define a finite domain partition for each node of the schema tree S, we can show that the size of the conditional tree type is bound, in contrast to [ASV01], where no domain partitions are used. Assume that S has n schema nodes and each node s_i is partitioned into m_i domain ranges. Then in the worst case, where there is exactly one repeatable node s' under the root and s' is the only clonable node in S, the maximum number of specialized types is $\prod_{i=1}^{n} m_i$ and the maximum size in number of nodes of the conditional tree type τ is $n \cdot \prod_{i=1}^{n} m_i$, since each specialized type will have at most n nodes.

4 Remainder and Refinement Queries

Given a MIT $\mathbf{T}$, with conditional tree type τ, and a ps-query q, the remainder query q^R, which is expressed as a set S^R of *local* remainder queries, is equivalent to the query $q - \tau$, that is, q^R retrieves from V the data not stored in $\mathbf{T}$. The navigation of the XML view V starts from the nodes that correspond to the suitable leaf nodes of the data tree T_d of $\mathbf{T}$ and not from the root of V, to avoid navigating on nodes already in T_d. We call such queries *local ps-queries* [ASV01], which are ps-queries that operate on the subtree rooted at a node n different than the root of V. Local ps-queries are possible because for each node in T_d we keep a pointer to the corresponding node of V. For example, consider the MIT of Figure 2 and the normalized query $Q2^N$ corresponding to the conditional tree type τ^N_{Q2} in Figure 4(b). The remainder query $Q2^R$ is the local ps-query shown in Figure 5.

The local remainder queries are generated in PTIME by the following procedure: First the normalized conditional tree type τ^N_q is rewritten such that each specialized type refers to single domain ranges, that is, τ^N_q is split into *minimal* specialized types. These are either completely contained or disjoint to a specialized type of τ of $\mathbf{T}$. Then, each minimal specialized type of τ^N_q is checked for containment against τ. The specialized types of τ^N_q, which are not contained in τ, form a minimal and complete set of remainder queries for q. Finally, they are merged into a smaller number of local ps-queries and sent to the XML database server.

Due to the normalization process with domain ranges in the $RefineMIT$, the number of local queries is small, in contrast to [ASV01], by an analysis similar to the one about the maximum conditional tree type size in Section 3. This property considerably improves the performance because it avoids the overhead of executing many "small" queries on the XML query engine.

The results of the remainder queries arrive at the query composer, where they are filtered to remove any data retrieved due to the normalization of the range conditions. At the same time, the refinement query q^F extracts the data in the data tree T_d that are in the answer of q. It is:

$$q^F = q^E \cap \tau \tag{2}$$

The filtered results $q^{R_f}(V)$ of the remainder queries are merged with the results of q^F, and q^C is executed on their result. It is

$$q(V) = q^C(q^F(T_d) \cup q^{R_f}(V)) \tag{3}$$

5 Replacement Algorithm

The replacement algorithm selects one specialized type t of the conditional tree type τ of the MIT $\mathbf{T}$ and removes it along with the data in the data tree T_d of $\mathbf{T}$ that conform to t. The decision is based on the timestamps lists of the specialized types. We use LRU, although other more sophisticated approaches [RV00] can be employed. Only the leaf specialized types are candidates for removal, to retain the coherence of $\mathbf{T}$.

For example, the candidate specialized types for the conditional tree type of Figure 4(c) are the two seller specialized types. Both of them have the latest timestamp in their timestamps lists, since they overlap with $Q2$, so the replacement algorithm selects arbitrarily one of them. Without loss of generality, suppose that the second specialized type is selected. An update extended XQuery [TIHW01] q^P is generated that deletes the data that are relevant to the selected specialized type, and q^P is executed against T_d. In particular, q^P is:

`FOR` $\$p$ `IN` $document(\text{“}catalog.xml\text{”})/catalog/product$, $\$s$ `IN` $\$p/seller$
`WHERE` $\$p/cat = \text{“}cars\text{”}$ `AND` $\$s/price >= 8000$ `AND` $\$s/price < 9000$ `AND` $\$/rating > 4$
`UPDATE` $\$p$ { `DELETE` $\$s$ }

If the space emptied is not enough to insert the data of the new query to $\mathbf{T}$, another specialized type is selected and removed in the same way.

6 Conclusions and Future Work

We presented a system for the efficient caching of a broad class of XML queries on XML databases, where the cache is organized as a MIT. We are working on implementing the system to measure the improvement in the response time for a query workload and experiment with replacement policies other than LRU. Also, we are working on an algorithm, that will be executed periodically on the incomplete tree, to compact the conditional tree type's representation. Finally, we work on the pipelining opportunities that the system presents in order to increase its throughput.

7 Acknowledgements

We wish to thank Yannis Papakonstantinou for the ideas he proposed and Victor Vianu for the discussions we made regarding the properties of the incomplete tree.

References

[ASV01] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and Querying XML with Incomplete Information. In *Symposium on Principles of Database Systems*, 2001.

[DFJ+96] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *The VLDB Journal*, pages 330–341, 1996.

[GG99] Parke Godfrey and Jarek Gryz. Answering queries by semantic caches. In *Database and Expert Systems Applications*, pages 485–498, 1999.

[LC99] Dongwon Lee and Wesley W. Chu. Semantic caching via query matching for web sources. In *CIKM*, pages 77–85, 1999.

[LN01] Qiong Luo and Jeffrey F. Naughton. Form-based proxy caching for database-backed web sites. In *The VLDB Journal*, pages 191–200, 2001.

[RV00] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[Sel88] T. Sellis. Intelligent caching and indexing techniques for relational database systems, 1988.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *SIGMOD Conference*, 2001.

[W3C01] W3C. XQuery: A query language for XML, 2001. W3C Working Draft available at http://www.w3c.org/XML/Query.