Efficient Evaluation of Preference Query Processes Using Twig Caches

SungRan Cho
L3S Research Center
Hannover, Germany
scho@L3S.de

Wolf-Tilo Balke
L3S Research Center
Hannover, Germany
balke@L3S.de

Abstract— **Facing today's information flood needs efficient means for personalization. Therefore XML query processing over large volumes of data needs to make the most out of already spent processing time by caching common (sub)expressions for reuse. This is especially promising for the new paradigm of personalized preference queries. Here a sequence of possible query relaxations is a-priori determined by the users' preferences. Structural and value-based preferences thus define a query process where predicates are progressively relaxed until a suitable set of best possible results has been retrieved. To improve evaluation times for such query processes we argue that caching intermediate join results of structural preference queries is especially effective, because subsequent queries will always be subsumed by some previously cached queries to a certain extent. In this paper we propose a structural join-based caching scheme that allows preference queries to reuse the most beneficial structural join results of all previous queries. We first design a twig cache along with effective strategies for cache management. Moreover, we present a selection algorithm for join orders using cached data and the preference-induced sequence of future queries to select optimal query evaluation plans. Our benchmark experiments show that by using our twig caches preference query processing can be essentially sped up.**

Keywords- XML databases, query processing, preference-based retrieval, caching

I. INTRODUCTION

XML is the standard representation and exchange format for a wide variety of frequently accessed information. A large amount of research in XML query processing is therefore invested to improve retrieval times over ever-growing volumes of data. To make the most out of already spent processing time sophisticated personalization techniques do not only consider individual queries, but cache common (sub)expressions. Since XML conforms to a labeled tree, basic building blocks for queries in *XPath* or *XQuery* are regular path expressions and tree patterns. Caching this basic structure for reuse has been shown to result in significant query time improvements. Caching approaches for XML documents have studied caching frequent query patterns based on statistical considerations [1] and caching on semantic considerations [2].

For effective caching the main problem is predicting future queries with high accuracy, and then choosing appropriate cache replacement policies to achieve a maximum of reusability of recent queries or fragments. But for preference queries, which are needed in a lot of practical applications such as e-business scenarios (see, e.g., [12, 14]), such a prediction is different: by user provided relaxation schemes each individual query is extended to a query process, i.e., a predetermined sequence of queries to be posed against the database. The exact number of queries from each sequence that will actually be posed depends on user satisfaction: the fewer the results are returned by the initial queries of the sequence, the more the query has to be relaxed and the more queries from the sequence will have to be posed to satisfy the user. Thus, in contrast to previous approaches an effective caching strategy supporting current users will not depend on long term statistics or semantic patterns, but rather on the users' preferred relaxation patterns and the structure they imprint on short term query processes.

Based on these insights, in this paper we develop a twig caching strategy for structural preferences in XML queries. Besides the basic benefits that can be expected of a fragment's reuse normalized by the cache space it needs, the design of our twig cache's profit model also weighs the benefit of a fragment to further queries in the sequence. To summarize, we make the following contributions:

- We propose a twig cache, a structural join-based caching scheme that allows preference queries to reuse the most beneficial structural join results of all previous cached queries. We also develop incremental maintenance strategies that are fundamental to such a twig cache. Our benefit model for caching is based on the notion of distance between the current query and a query later in the sequence: realizing a gain for a near query that very probably will have to be posed, is better than gains only being realized in later queries, that might probably not be posed at all (Section 3).
- Since all possibly cacheable fragments depend on the chosen execution plans, we design a greedy heuristic algorithm for selecting best join orders. Considering each query and the currently cached data, our algorithm generates a most profitable evaluation plan by identifying the set of maximally beneficial structural join results for not only the current query, but also future queries in the sequence (Section 4 and 5).

- We implemented our twig caching strategies and ran a thorough performance evaluation based on the XMark benchmark datasets. Our experimental results show that for evaluating a given set of preference queries our sophisticated twig cache generally achieves a substantial performance gain as compared to both non-caching and the often employed LRU replacement strategy (Section 6).

II. QUERIES WITH STRUCTURAL PREFERENCES

Given today's enormous volumes of information, retrieval has to utilize ranking-based approaches allowing a maximum of personalization. In ranking most relevant information users generally express certain preferences in their queries. Such preferences state what a user likes/dislikes and preference query processing attempts to retrieve all best possible matches in a cooperative fashion avoiding empty result sets. Whereas in relational databases preferences are only expressed on values of certain predicates, for semi-structured data the preferences in queries can also be structural. Since the structure in XML carries a certain semantics users can specify a preferred path to the actual information that can then be relaxed step by step to find only most relevant matches.

Extending the model of preferences as strict partial orders with an easy use of *'I like A better than B'* semantics (see, e.g., [12, 14]), we define structural preferences for queries as:

DEFINITION 2.1. [Structural Preference Graph] A *structural preference graph* is a finite node-labeled directed graph P = (V,E), such that: (i) each node in V is labeled by a preference value, (ii) the direction of each edge between nodes in P expresses that the attribute of the label of the node where the edge originates is preferred over the attribute of the label to which the edge points.

For instance, Figure 1(a) shows two structural preference graphs P_{type} and P_{search}, in which P_{type} graph expresses that the structural element *sedan* is preferred over *SUV*, whereas the P_{search} graph states that *price* is preferred over *location*. To support more than a single structural preference, in the following we use a fair relaxation scheme based on *Pareto optimality* principle. For example, Figure 1(b) shows the Pareto order induced by the preferences P_{type} and P_{search} in Figure 1(a). The sedan, price combination is obviously the most preferred option; next are the equally preferred *sedan*, *location* and the *SUV*, *price* options such that both relax one preference each; the least preferred option is any combination of structural items (entirely dropping both user preferences).

To use structural preferences in XML queries some leaf nodes of a query can be marked with preferences:

DEFINITION 2.2. [Preference Query] A *preference query* is a rooted node-labeled and edge-labeled tree such that (i) its nodes are labeled with element tags, with the exception that its leaves may be labeled by element tags or values, (ii) its edges are labeled parent-child or ancestor-descendant, (iii) some element nodes are marked as distinguished and subsequently returned as query result, and (iv) some element leaves are marked with preferences.

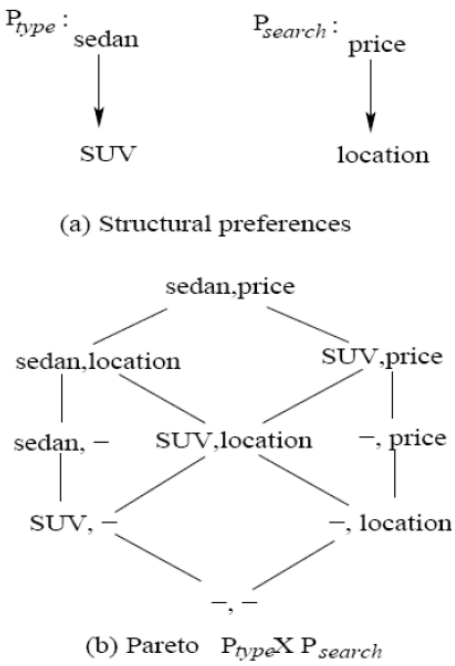


Figure 1. Example of a Structural Preference

For example, consider a user who wants to buy a car and do some research on new cars before making a decision. Of course, the user may have preferences about the car type and/or search parameters, as given for instance in Figure 1(a). The query *Q* in Figure 2 shows the appropriate preference query the user requests, where the query nodes *new-car* and *offer* are marked with the preferences P_{type} and P_{search}, respect tively. Please note that all items shown here are element tags, but they could of course be mixed with values. Generally edges are either parent-child (single edge) or ancestor-descendant (double edge). In each query, some nodes are marked with '*' indicating that these nodes are returned as answer (i.e., projected).

For the actual evaluation we can incorporate the user's preferences into the following XQuery path expression of query *Q* in Figure 2 with the preferences:

sync(e-car/buy[.//new-car[preference(P_{type})]]
//offer[preference(P_{search})])

In our evaluation framework a preference is implemented by adding a predicate containing a preference function with P_{type} as an input argument to a query node. In the unfolding process the function expands the marked query node by appropriate attributes in P_{type} and P_{search}. For evaluation a preference query needs to be rewritten into an *ordered sequence* of queries. We have to consider all possible relaxations of the user preferences in the user specified order by expanding marked nodes accordingly. Our framework's *sync* function interacts with multiple preference functions given in the query and computes their conjunction following the Pareto semantics. Next we formalize how the *sync* function unfolds preference queries for query processing in case of a single preference in a query:

Identify applicable sponsor/s here. If no sponsors, delete this text box. (sponsors)

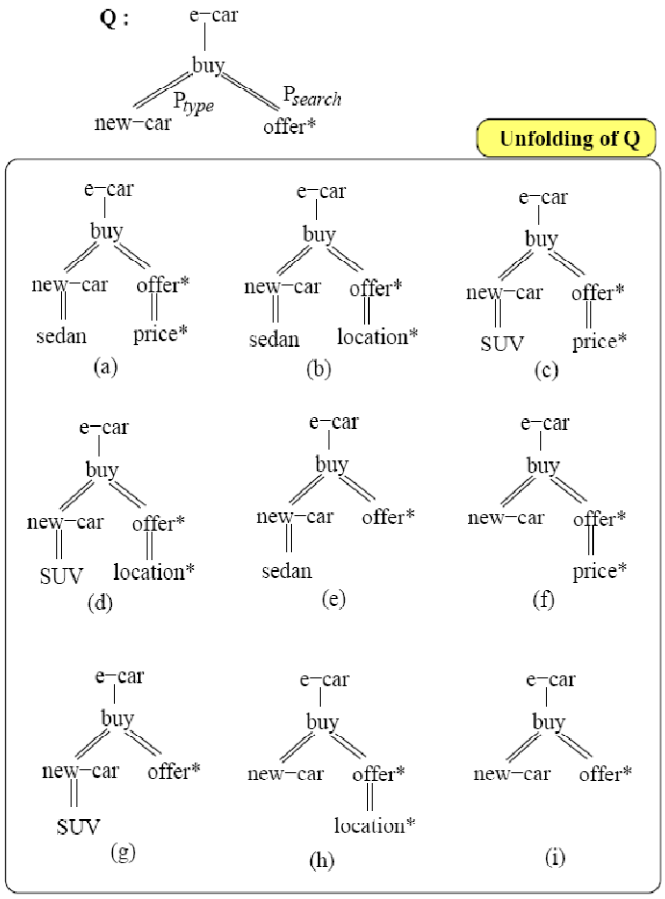


Figure 2. Unfolding a Query

DEFINITION 2.3. [Unfolding Structural Preference Queries] Given a query Q containing a node n marked with a structural preference P, an expansion query Q' is obtained by *unfolding* Q in the following way:
(i) while respecting the order induced by P,
- adding a node v of P to a query node n as a successor (in case that n is value-constrained, v resides between n and the value),
- adding an edge with ancestor-descendent label from n to v,
- propagating a possible distinguished status of n down to v.

(ii) entirely removing P from Q.

Whenever preference marked nodes are distinguished nodes, their expanded nodes inherit the status of projection. The intuition behind this is that answers should contain the projected nodes, but should also show in how far the user's preferred nodes have been granted. For this reason, the queries in Figure 2(a)-(i) additionally project *price* and *location* nodes under *offer* node.

The generalization of definition 2.3 to the case of multiple nodes marked with preferences is straightforward using the Pareto optimality principle: each single preference is expanded individually such that every possible combination of preference attributes is reflected by some query expansion. If the query contains multiple preferences $P_1, \ldots, P_n$ and each preference P_i consists of $|P_i|$ distinct attributes, the total number of queries posed will be $|P_1 + 1| \times \cdots \times |P_n + 1|$. Given the query Q in Figure 2 containing the two structural preferences P_{type} and P_{search} from Figure 1(a), the induced Pareto order is reflected by 9 possible expansions of Q in Figure 2(a)-(i) in the unfolding process.

Since all expansion queries are conventional XML queries, they can be posed against any XPath or XQuery evaluation engine. In the context of XML, several query evaluation strategies (e.g., cost-based optimization or choosing special algorithms for specific structural joins and specific indexes) for a query have been extensively studied. In particular, structural join operations are essential to evaluate XML queries over databases and query evaluation plans basically make use of two binary structural join predicates, i.e., the parent-child and the ancestor-descendant relationships. In this paper we will use stack-based structural join algorithms, namely Stack-Tree, which form a class of efficient and widely-used algorithms for computing binary structural joins [3, 5].

In a nutshell structural join algorithms work as follows: all nodes in tree-structured XML data are encoded with preorder, postorder, and level for ordered and structural matches. The algorithm takes input datasets ordered by a node's preorder and outputs the results of the structural join ordered by either ancestor (or parent) or descendant (or child) participating in the join, by using in-memory stacks.

III. DESIGNING A TWIG CACHE

In this section we first present the problem of preference-based structural XML retrieval and motivate our solution. Next, we design a twig cache and consequently focus on incremental admission to, replacement in, and removal from the twig cache, in order to obtain highest cache usage in handling a set of queries expanded by user's preferences.

We investigate caching techniques that improve query response times to answer queries rewritten to reflect user's preferences not only in the case where all queries are evaluated, but also in the case where *only some* queries have to be answered:

Preference Query Evaluation Problem: Given a set of queries expanded by user's preferences, $Q_1, \ldots, Q_n$, and an XML document D, retrieve $R_1, \ldots, R_n$ with *improved response time*, where R_i is the result of Q_i over D.

The following observation describes a key feature of preference queries, which motivates our solution:

Locality of Preference Queries: Preference queries typically feature repetitive fragments and always follow induced patterns. For example, consider the expansion query in Figure 2(a) from the unfolding of Q. A subtree of the query, *pc-relationship(e-car,buy)*, is a part of its all subsequent queries, i.e., Figure 2(b)-(i). Such locality is a consequence of the step-wise relaxation process.

One possible way to process preference queries is to apply multi-query evaluation techniques exploiting common subexpressions. However, then all queries will have to be posed. Since we do not a-priori know how many of the queries will be necessary to satisfy a user's information needs, we propose a *twig cache* to address the problem. The twig cache

stores results of structural joins before they are referenced in anticipation of a miss (concurring to the notion of *prefetching*).

A. Building a TwigCache

Caching has been traditionally done at query level, i.e., complete query results are stored in the cache. Any new query can then be answered using the cache by detecting query containment. In order to support query processes, however, we will not store monolithic result sets of queries that could only be reused if exactly the same query is repeated. Instead we decompose the query into structural join operations and store intermediate join results in a twig cache such that a subsequent query can be partially or even entirely evaluated using the cache. Hence, a twig cache contains a set of structural join results along with meta-data that succinctly provides the information about the data cached. For the use in evaluating preference queries our twig cache only stores most beneficial structural join results to improve the total evaluation time of the induced sequence of queries. We formalize this as follows:

DEFINITION 3.1. [Twig Cache] A twig cache for a preference query is defined as a 5-tuple:

TC = (SJ, Time, Size, Ref, Instance),

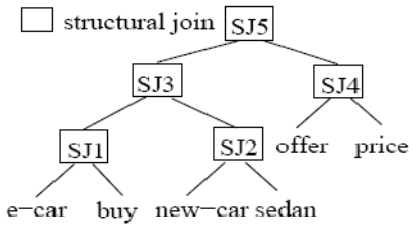
where SJ is a structural join identifier, Time is the execution time spent to perform the structural join, Size is the total size of the structural join result, Ref is a list of future queries from some query process that reference the structural join, and Instance is the cached instance results

In the twig cache, *Time* is set to the actual time needed to execute the structural join, whose results are cached. But higher-up joins making use of previously produced join results set *Time* as an accumulation of its time and the time needed for all base joins.

In our join-based twig caching framework, it is essential to develop novel strategies for admitting and replacing join results in the cache to make effective use of the limited cache space and avoid the recomputation of particularly expensive structural joins. Following the notion of prefetching we generally try to materialize join results that will be referenced in near future. If, however, the result of a structural join does not fit into the cache, our benefit model identifies those candidates with least benefits and evicts them from the cache to free up enough space. All twig cache operations (admission, replacement, and removal) determine beneficial candidates in the cache by taking advantage of our meta-data information.

Twig cache admission and removal strategies are incrementally performed by associating a benefit with each join candidate. Potentially beneficial structural join candidates for future queries are materialized in the cache; redundant joins or joins no longer beneficial are eliminated from the cache.

Consider the evaluation plan in Figure 3(a) for the first query (a) in the query process of our sample unfolding in Figure 2. A naive caching scheme would store all the intermediate results, SJ_1 - SJ_5. However, if we take a closer look at all the joins (SJ_1 - SJ_5), it turns out that the result of SJ_5 does not contribute to any future query within the query process (i.e., its benefit is 0). Thus, SJ_5 should definitely not be admitted into the twig cache. More generally, the last join in any evaluation plan will never be referenced by following queries in the course of a query process. Thus, it should not be admitted to the cache in the first place.



(a) An evaluation plan of Q(a)

Structural Join	Time(sec.)	Size(KB)	Ref
SJ1(e−car,buy)	2	25	b,c,d,e,f,g,h,i
~~SJ2(new−car,sedan)~~	1.5	20	b,e
SJ3(SJ1,SJ2)	3.2	30	b,e
SJ4(offer,price)	1	15	c,f
~~SJ5(SJ3,SJ4)~~	3.5	45	

(b) Twig cache after evaluating Q(a)

Figure 3. Evaluating Q(a)

Similarly a cached join result s_1 should be removed from the twig cache whenever one of the following holds: (i) if s_1 is base data for a higher-up join s_2 in the evaluation plan and s_1, s_2 are referenced by the same queries; (ii) if s_1 is no longer referenced by any following query. For instance, from the evaluation plan of Figure 3(a) and the meta-data of the twig cache in Figure 3(b), we can see that the join SJ_3 is produced by using the result of SJ_2 and both joins are referenced by the same future queries (b) and (e) in Figure 2. Because referencing SJ_3 is sufficient to evaluate both queries, we can remove the low benefit join SJ_2 from the twig cache. Finally, the cache only contains the join results SJ_1, SJ_3, and SJ_4, that are used by all following queries.

Cache replacement strategies generally play a central role in managing caches, since cache sizes are always restricted. We make use of a strategy based on the execution time, size of a query, and the probability of reference in the future for deciding which data to cache (see, e.g., [7]). However, our twig cache replacement strategy also has to take into account the locality property of a preference query and the different benefits of joins at different levels in the evaluation plan. Incorporating this into the model, our cached benefit of a structural join *sj* is computed as

CacheBenefit(*sj*):

$$\frac{\sum_{i=1}^{numRef} Time(sj) \times \frac{1}{distance_i+1}}{Size(sj)}$$

In this benefit model, the cached benefit of each structural join is obtained as the cumulated time needed to execute the

join, normalized by the distance until it will be referenced and its size in the cache. The parameters *Time*, *Size*, and *numRef* (the number of referencing queries) are provided in the meta-data of the twig cache. *Distance* measures the closeness of referencing queries by computing the distance between the current query and the next referencing query in the query process. Distances for equally preferred queries are considered to be equal, due to the fact that their order to be evaluated is arbitrary. For example, Figure 2(d),(e), and (f) queries all have a distance of 3 from query (a), because they all perform two relaxation steps on base preferences (cf. Figure 1(b)).

Twig cache replacement is now performed by computing the benefit of each structural join in the cache to find candidates to be evicted. The benefits of the candidates should be lower than that of the new join and after evicting the size of the new join should fit into the cache. Whenever a new structural join is stored in the cache, its associated time, size, and reference queries are recorded as associated meta-data. Moreover, whenever a query is completed, the contents of *Ref* in the meta-data set of the cache are updated accordingly so that the current benefits can be taken into account during cache replacement and removal operations.

IV. SELECTING QUERY EVALUATION PLANS

In this section we develop techniques for selecting the best evaluation plan of not only the current query, but also the subsequent queries of its query process. Our technique takes advantage of cached joins and determines highly beneficial join results to be cached for future reference. With an adequate model a good decision for choosing an evaluation plan of a query can be made by computing the estimated profits of all possible alternative plans. We will first define the cost of a structural join *sj* (i.e., the cost of the Stack-Tree algorithm), where *A* is ancestor of *B* in the query, as follows:

Cost(*sj*):

$$| A | \times f_{stack} + | A | \times f_I + | B | \times f_I + | AB | \times f_O \quad (1)$$

In the cost model, |*A*| is the cardinality of *A* and |*AB*| is the cardinality of the join result of *A* and *B*. Besides, it uses f_{stack} for the cost of a stack operation, f_I for a cached/(indexed) disk input access operation, and f_O for a cached/disk output operation to normalize the cost. f_I and f_O return 0 for cached I/O. If a join is beneficial in the future, its result is considered to be output into the cache; otherwise, output to disk. The cost of sorting the results, f_{sort}, may be additionally required.

We adopt the profit metric introduced in [6], considering a KNAPSACK problem that selects elements in decreasing order of benefit/size. We then slightly customize the model to suit our application. As a main factor of our profit model, we thus formalize the total benefit of a structural join *sj* for a set of relevant queries as follows:

TotalBenefit(*sj*):

$$\sum_{i=1}^{numRef} \frac{Cost(sj)}{distance_i + 1}$$

As introduced in Section 3.4, distance is a factor to capture the closeness of reference queries and *numRef* is the total number of queries referencing to *sj*. The total benefit of *sj* is computed as the sum of Cost(*sj*) with (indexed) disk input and cache output costs in the period of referencing *sj*. Finally the profit model of a structural join *sj* of *A* and *B* (|*AB*| is the size of the join result of *A* and *B*) can be computed as:

Profit(*sj*):

$$\frac{TotalBenefit(sj) - Cost(sj)}{| AB |}$$

Our profit model not only captures the benefit of a cached join, but also assesses the estimated benefit of a cached join result in the future. The profit model is used for meeting the following two key points for a set of queries: (i) what is the most profitable evaluation plan of the current query to maximize the usage of cached data; (ii) what is the most profitable evaluation plan of queries evaluated later in the query process allowing to maximize future contributions.

Our models immediately cater for the case where only a few best answers are requested: the distance function elegantly captures a key feature when the first few queries in a sequence can probably already meet the user's information needs.

A. Selecting Query Evaluation Plans

In order to efficiently evaluate a preference query, our approach focuses on (i) analyzing the structure of all possible queries in the unfolding; (ii) dynamically translating each query into the most profitable evaluation plan by incorporating cached joins; (iii) efficiently managing the twig cache to make effective cache utilization; (iv) evaluating the query evaluation plan selected and finally return answers.

In XML databases, the selection of join order is generally an important task. In particular, structural join operations, which focus on ancestor-descendant or parent-child relationships of the XML elements to be joined, are central to evaluating queries against XML databases. Several algorithms for structural join order optimization have been proposed (see, e.g., [4]). However, conventional join order selection algorithms finding the cheapest plan focus only on a single query.

Since our focus is to efficiently evaluate a given set of queries, we develop an algorithm for selecting join orders of a sequence of queries in a given query process. The key ideas behind our algorithm are: (i) to account for cached data as early as possible; (ii) to obtain a fair caching of structural joins; and (iii) to make an effective utilization of the twig cache.

Greedy Heuristic Algorithm: This algorithm generates the most profitable evaluation plan of a query by locally selecting largest profit joins at each join step. The algorithm for selecting join orders works as follows: it first identifies cached joins and then decides necessary joins of the query based on their profits by traversing the query in top-down fashion. In order to generate higher-up joins, it picks the join *s* with largest profit from all generated base joins, and by comparing the profits between *s* and each of the other join candidates a

higher-up join of *s* (which again has the largest profit) is determined. This process continues by picking the next highest profit join *s*. We repeat this iteratively until all query nodes participate in the plan. The algorithm is presented in detail in Figure 4.

```
Algorithm MPP
Input:  query Q;
Output:  a most profitable evaluation plan;
/* find-cachedjoins() reads twig cache and Q,
  marks query nodes if there are relevant
  cached joins.  */
find-cachedjoins(Q);
navigate-query(Q.root);
bottom-up(ep);
end Algorithm

procedure navigate-query
Input:  node n;
Output:  a list of joins ep;
if (!n.visited) {
  profit = 0;
  if (n does not have a child) cnode=n;
  for each child c of n {
    if (!c.visited & profit < getprofit(n,c)) {
     profit=getprofit(n,c);
     cnode=c;}}
  generate-plan(n,cnode,ep);
  n.visited = cnode.visited = true;}
for each child c of n
  navigate-query(c);
end procedure

procedure bottom-up
Input:  a list of joins ep;
Output:  a list of joins uep;
/* isPath(p,q) returns TRUE, if there is a query path
  between a node in a list p and a node in a list q;
  otherwise return FALSE. */
/* get-highestprofit() returns an unvisited join
  with largest profit.  */
while (there is an unvisited join in ep) {
  profitjoin = get-highestprofit(ep);
  profit=0; joinstatus=false;
  for each unvisited join p(≠profitjoin) of ep {
    if (isPath(profitjoin,p)) {
      if (profit < getprofit(profitjoin,p)) {
       profit=getprofit(profitjoin,p);
       tempjoin=p;}
      joinstatus=true;}}
  if (!joinstatus) tempjoin=profitjoin;
  generate-plan(profitjoin,tempjoin,uep);
  profitjoin.visited = tempjoin.visited = true;}
if (uep.size() > 1) bottom-up(uep);
end procedure
```

Figure 4. Algorithm MPP

The algorithm focuses on more promising evaluation plans of the search space and helps to reduce the overhead associated with frequent cache replacement. It is clear that cached data generally has higher profits and if low benefit joins are early admitted into the cache, then these will be candidates to be replaced by upcoming higher benefit joins.

To demonstrate how the algorithm works, consider the query in Figure 2(a) and start with an empty twig cache. We traverse the query in top-down fashion and determine largest profit joins based on local decisions. The root *e-car* and its child buy are first joined (SJ_1). Similarly, *new-car* and *sedan* nodes are determined to be joined (SJ_2) and the *offer* and *price* nodes form another join (SJ_4). To determine the next higher-up joins we scan the base joins produced: Starting with the join with largest profit, namely SJ_1, we compute the join profits between SJ_1 and SJ_2, and between SJ_1 and SJ_4. The algorithm decides for joining SJ_1 and SJ_2, because the profit is larger. Lastly SJ_4 participates in the final evaluation plan. Figure 3(a) depicts our most profitable evaluation plan of the query. Please note that this plan may not be the cheapest plan from a traditional query optimization standpoint, but using our plan the intermediate structural joins SJ_1-SJ_4 can be further utilized by subsequent queries of the preference query process.

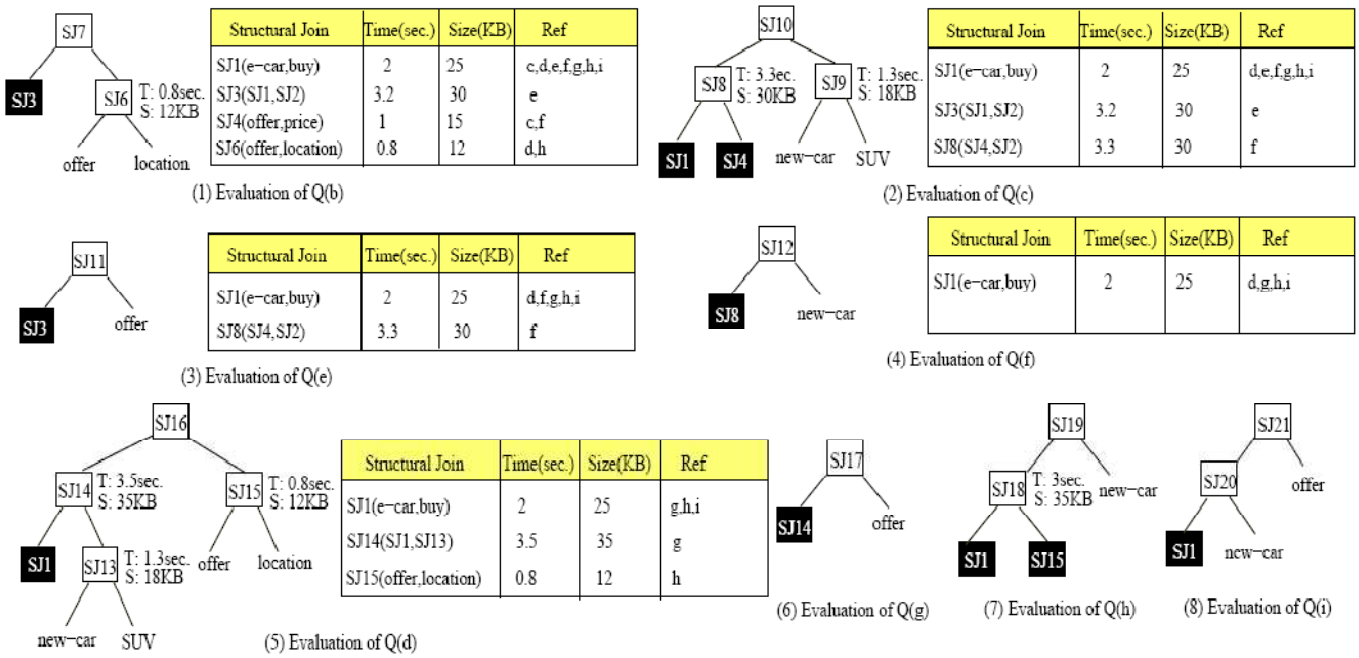
Before starting the evaluation of a set of queries, queries are grouped by their degree of relaxation (or for short 'distance') in the preference-induced order. For example, queries (a)-(i) in the unfolding shown in Figure 2, can be grouped as follows: [{(a)}, {(b), (c)}, {(d), (e), (f)}, {(g), (h)}, {(i)}]. Even though equally preferred queries do not have to be executed in a specific order, the cached data allows us to choose which of those queries should be evaluated next.

In the preference community, dealing with equally ranked preference attributes which are usually considered incomparable, frequently occurs in managing real world trade-offs. Even when each individual preference defines a total order, combining multiple preferences may result in incomparability such as our example with Pareto semantics in Figure 1(b). Although we cannot sensibly determine an order between equally preferred queries, we can nevertheless decide which of them takes benefits most from the current cache and thus should be evaluated first.

The cost of an equally preferred query q is computed as $\sum_{i=1}^{numSJ} Cost(i)$, where *numSJ* is the number of structural joins participating in a selected evaluation plan of q. This cost measures the benefit of current cached data for a query as the sum of the cost of each structural join (presented in Eq. 1) constituting an evaluation plan. The cheapest query plan that implies a largest cache benefit is selected to be evaluated next.

V. AN ILLUSTRATIVE EXAMPLE

For ease of understanding we illustrate all our techniques using our running example. Let us revisit the scenario in Figure 3 with the cache capacity of 1MB. Given the query in Figure 2(a), Figure 3(a) and (b) show its selected evaluation plan and the updated contents of the twig cache after evaluating the query, respectively. Having finished the first query, next we have to deal with the equally preferred queries in Figure 2(b) and (c).



Structural Join	Time(sec.)	Size(KB)	Ref
SJ1(e-car,buy)	2	25	c,d,e,f,g,h,i
SJ3(SJ1,SJ2)	3.2	30	e
SJ4(offer,price)	1	15	c,f
SJ6(offer,location)	0.8	12	d,h

Structural Join	Time(sec.)	Size(KB)	Ref
SJ1(e-car,buy)	2	25	d,e,f,g,h,i
SJ3(SJ1,SJ2)	3.2	30	e
SJ8(SJ4,SJ2)	3.3	30	f

Structural Join	Time(sec.)	Size(KB)	Ref
SJ1(e-car,buy)	2	25	d,f,g,h,i
SJ8(SJ4,SJ2)	3.3	30	f

Structural Join	Time(sec.)	Size(KB)	Ref
SJ1(e-car,buy)	2	25	d,g,h,i

Structural Join	Time(sec.)	Size(KB)	Ref
SJ1(e-car,buy)	2	25	g,h,i
SJ14(SJ1,SJ13)	3.5	35	g
SJ15(offer,location)	0.8	12	h

Figure 5. Evolution of the Twig Cache and Query Evaluation Plan Selection

We first produce evaluation plans with the largest profits of the two queries, shown in Figure 5(1) and (2), where cached joins are depicted within dark rectangles. For both plans, we compute their costs to identify which plan takes more advantage of the cache. Since the query in Figure 2(b) draws more benefit from the cached data, we pose it next. While evaluating the query plan, a structural join SJ_6*(offer,location)* is admitted into the cache, but join SJ_7 is discarded, because it is not beneficialfor future queries. After finishing the query, the meta-data ofthe cache is updated for the next round as shown in Figure 5(1), i.e., in the *Ref* field, the current query (b) is simply deleted from SJ_1 and SJ_3 joins. In general, when the cache is newly updated, evaluation plans of the next queries should be reproduced in order to always reflect the current status of the cache.

In Figure 5(2) we can use the same evaluation plan, but need to decide whether SJ_8 should be admitted into the cache. Since the total benefit of the two joins SJ_4 and SJ_6 is lower than that of SJ_8, they are replaced. In contrast, the join result of SJ_9, whose benefit is lower than any cached join, is not admitted.

Next, the queries in Figure 2(d), (e), and (f) are considered together and the evaluation plans selected are shown in Figure 5(3), (4), and (5), respectively. For these equally preferred queries, we now select an evaluation plan taking the largest cache benefit. The query in Figure 2(e) thus is chosen as first query. While evaluating this query, no new join result is admitted into the cache, but the result of SJ_3 whose life span ends here, is eliminated, as shown in Figure 5(3). In the same manner the query in Figure 2(f) is selected next and Figure 5(4) shows the updated cache contents. While evaluating the last query in Figure 2(d), the new join results SJ_{14} and SJ_{15} are admitted into the cache, but at the same time the join result of SJ_{13} is discarded. This is because SJ_{13} and SJ_{14} are referenced by the same query in Figure 2(g) and the result of SJ_{14} is generated using the result of SJ_{13}.

When the next two queries in Figure 2(g) and (h) are considered, the query in Figure 2(g) whose evaluation plan is shown in Figure 5(6), is chosen to be evaluated first. Only SJ_{14} whose benefit is zero for subsequent queries is removed from the twig cache. From the query in Figure 2(h) and its evaluation plan in Figure 5(7), the intermediate result of SJ_{18} is checked into and out of the cache, and SJ_{15} that no longer can be used, is deleted from the cache. After executing the selected evaluation plan in Figure 5(8) of the last query in Figure 2(i), the twig cache is cleaned up.

VI. EXPERIMENTAL RESULTS

We ran extensive experiments using the XMark benchmark dataset. Our experiments were run on a SUN UltraSPARC-III (750 MHZ CPU). In our experiments, we used data (50MB) and a twig cache with a size of 2% of the data size. We compared the twig cache against non-caching and two different replacement strategies, our benefit-based replacement and the common LRU (least recently used) strategy. Note that all the evaluation times we report include the time spent on cache management and query plan generation.

We first ran a simple query and gradually increased query nodes with and without preference constraints in order to show the benefits of our twig cache. We constructed a simple branch query in which the query size is 4 nodes and two leaf nodes are marked with preference constraints, each of which contains two attributes. As a result, the number of possible expansions of the query is 9.

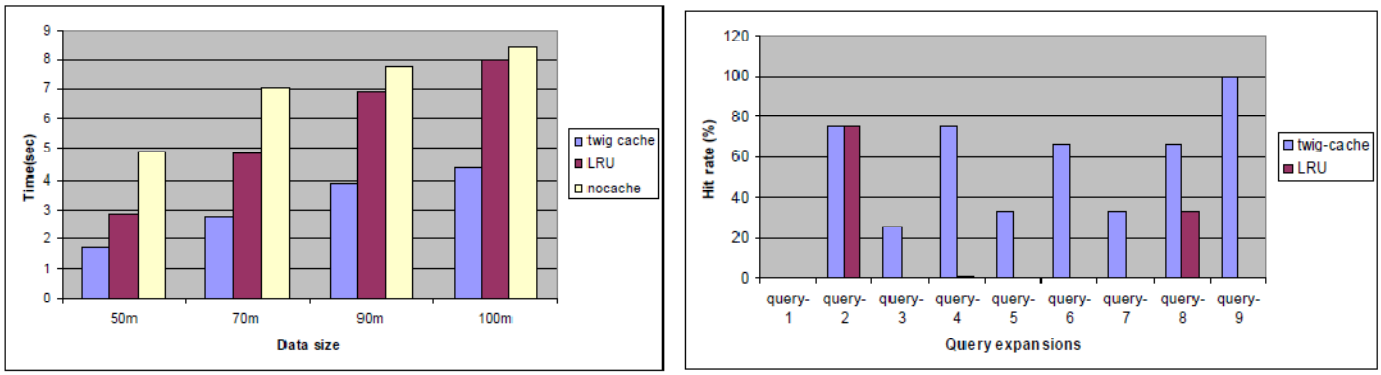


Figure 6. Varying Data Size (left) and Cache Hit Rates (right)

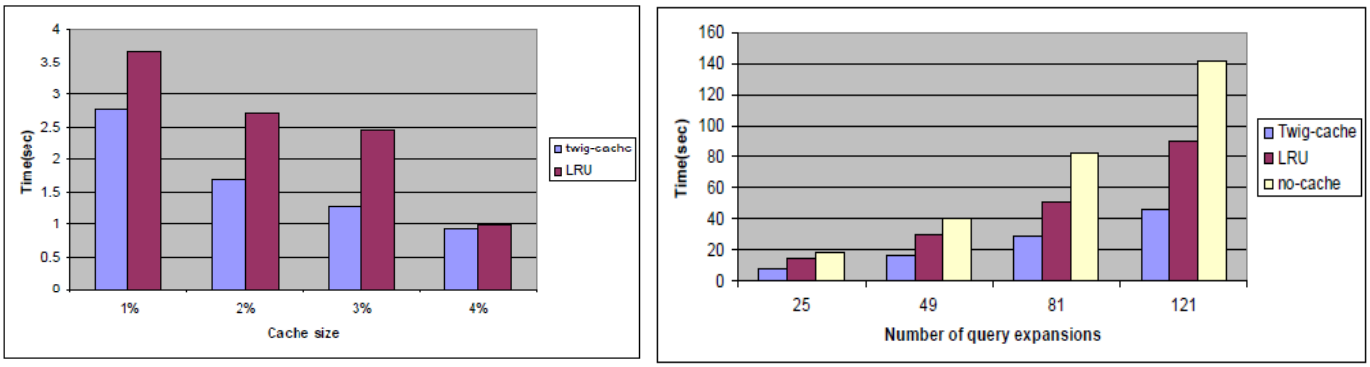


Figure 7. Varying Cache Size (left) and Preference Size (right)

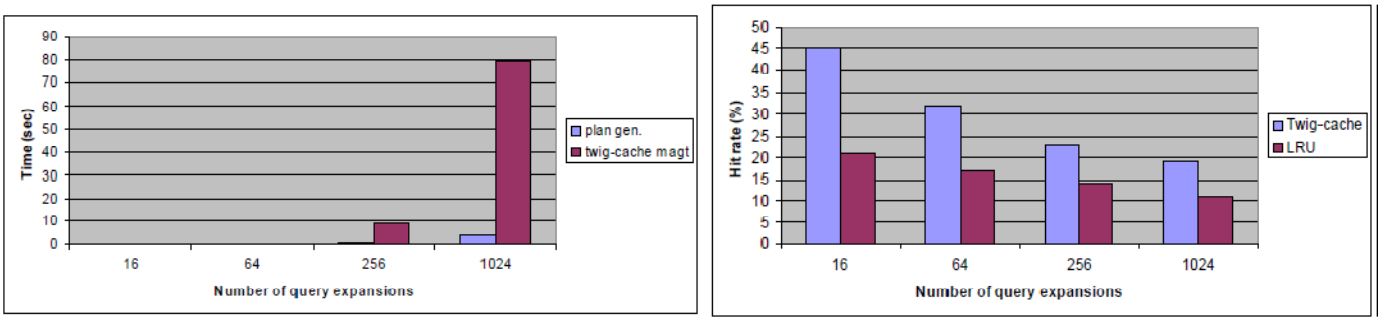


Figure 8. Studying Optimization Times (left)and Hit Rates with Varying Preference Constraints(right)

Figure 6 (left) compares cached and non-cache evaluation times (reported in seconds) of the query with varying dataset sizes from 50MB to 100MB, where cached times are divided into benefit-based replacement (described as twig-cache) and LRU strategies. The total evaluation time is obtained by adding up evaluation times of all expansion queries. The cached evaluation time of the query is about 3 times cheaper than the non-caching one. Additionally, using the twig cache the benefit-based replacement strategy is approximately 2 times cheaper than simple LRU.

Our next experiment more clearly demonstrates the superiority of our benefit-based replacement strategy over LRU. Figure 6 (right) reports the average hit rate of our 9 query expansions from the first experiment. The hit rate measures the ratio of cached join hits to the total number of joins in a query evaluation plan. After the first query, all following expansion queries result in cache hits in our benefit-based cache replacement strategy, while in the LRU strategy, 5 out of 8 queries do not obtain any cache benefit accounting for the worse evaluation times. We also measured the effect of the cache size on query evaluation. We varied the size of our twig cache from 1% of the data size to 4%, while keeping the data size fixed at 50 MB. Running the query in the first experiment the benefit-based replacement strategy outperforms LRU by about a factor of 2 for the cache size of 3%. As could be expected, with increasing cache size the benefit-based replacement and LRU strategies perform similar, because the cache stores almost all structural joins and less cache replacements occur. This trend is nearly the same with a large number of query expansions. Figure 7 (left) reports this result.

We also studied the costs of generating query evaluation plan and managing the twig cache. In order to understand this better, we ran more complex queries. We varied preference marked nodes in the query and measured optimization times

when the numbers of possible query expansions are 16, 64, 256, and 1024. The graph of Figure 8 (left) reports the cumulative times of plan generation and twig cache replacement for all query expansions. Since the graph does not display some plots, we here report the actual times: the times of plan generation are 0.008sec and 0.06sec when query expansions are 16 and 64, respectively and the times of twig cache management are 0.004sec (16 query expansions) and 0.1sec (64 query expansions). In fact optimization times are very low, approximately 5% of the total evaluation time even with a large number of queries.

Next we measured the hit rates from the previous experiment. The graph in Figure 8 (right) shows that cache hit rates in the benefit-based cache replacement strategy are always better than those in LRU. However, as the number of query expansions increases, the cache hits in both the benefit-based replacement and LRU strategies slowly decrease due to the fact that with limited cache size cache replacements occur frequently to evaluate a growing number of expansion queries.

In order to measure evaluation times of preference queries with different complexities, we built preference constraints by increasing the number of preferred attributes. Figure 7 (right) shows that the cached evaluation time of the query is constantly about 3 times cheaper than the non-caching one for growing sizes of preference constraints. Additionally the benefit-based replacement strategy is constantly about 2 times better than the LRU. The reason for this is that the twig cache effectively stores structural joins that can be possibly reused in near future.

VII. Related Work

Our work is closely related to previous work on caching and preference query processing. There has been significant research on these areas to improve overall query response times. We present related work within the following two categories.

Caching and reusing: Caching of query results has been studied extensively in the literature. In the context of data warehousing and OLAP, Deshpande et al. [9] proposed "chunking" architectures, in which the database is partitioned into groups of tuples to cluster together. Scheuermann et al. [7] showed cache replacement and admission schemes based on a query-specific profit metric. Dar et al. [10] studied semantic query caching for client-server architectures, where the cached data is divided into semantic data regions and a query is split into subqueries by identifying their intersection with cached regions. Among other approaches, data prefetching has been studied to decrease expected response time of anticipated requests (see, e.g., [8]). However, to the best of our knowledge, ours is the first work to consider structural joins of a query as the basis for prefetching data into twig caches.

Preference Queries: Recently, several systems for supporting preference queries have been proposed (e.g., see [12, 13, 14, 15, 16, 19]). Several proposals for supporting preference queries have focused on extending the standard SQL in a way that preference relation is embedded into traditional relational query languages to enable to select preferred data. Work by [13, 15] proposed preference XML queries in connection with the modeling framework for preferences as partial order graphs given in [12, 14]. In particular, in [15], the authors proposed the declarative language *PreferenceXPath* by extending XML queries using the XPath standard. The resulting language enables the use of soft filtering conditions for node-selection in XPath. Like our approach a soft condition defines a strict partial order over the set of elements to be filtered and then returns only the best matches.

Since an exact match retrieval model is often inadequate in practical scenarios, XML query engines have been extended to support an IR-stylematching of data values. Engines like XIRQL [17] and XXL [18] applied probability-based keyword retrieval to structured XML documents. Though these engines allow to find similar values for a certain predicate or relax query predicates to find desired values in related structural elements, they do not support structural user preferences handled by our approach.

VIII. Conclusions

Preferences are becoming a more and more important paradigm in XML query processing, allowing for a high degree of personalization with respect to a user's information needs. In this paper we have developed a framework for join-based twig caching for XML preference queries.

While the spirit of a twig cache to evaluate a set of expansion queries is similar to caching schemes proposed for different kinds of databases, there is a crucial difference in the necessity of relating already cached data to the selection of a query evaluation plan for incoming queries. By considering the special features of preference XML queries, we presented a profit model as the base of identifying potentially beneficial structural joins that can be reused at a later stage of a query process. To facilitate the intelligent generation of query evaluation plans we then presented a greedy heuristic algorithm that selects plans by not only considering benefits from currently cached joins, but also choosing joins whose results will be useful to answer the remaining queries of the query process. Our experiments showed the essential benefits of our approach for evaluating an anticipated set of queries.

We are currently working on incorporating our improved evaluation techniques into an optimization framework for complex query processes to improve overall performance for XML preference queries.

Acknowledgment

Parts of this work was carried out within the Emmy Noether Program of Excellence of the German Research Foundation (DFG).

References

[1] L. Yang, M. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *Proceedings of VLDB*, 2003.

[2] L. Chen, E. Rundensteiner, and S. Wang. XCache – A semantic caching system for XML queries. In *Proceedings of SIGMOD*, 2002.

[3] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: efficient matching of XML query pattern. In *Proceedings of ICDE*, 2002.

[4] Y. Wu, J. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of ICDE*, 2003.

[5] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: optimal XML pattern matching. In *Proceedings of SIGMOD*, 2002.

[6] O. Kapitskaia, R. T. Ng, and D. Srivastava. Evolution and Revolution in LDAP Directory Caches. In *Proceedings of EDBT*, 2000.

[7] P. Scheuermann, J. Shin, and R. Vingralek. WATCHMAN: a data warehouse intelligent cache manager. In *Proceedings of VLDB*, 1996.

[8] P. Cao, E. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of SIGMETRICS*, 1995.

[9] P. M. Deshpande, K. Ramaswamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of SIGMOD*, 1998.

[10] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB*, 1996.

[11] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35-47, 1996.

[12] J. Chomicki. Querying with intrinsic preferences. In *Proceedings of EDBT*, 2002.

[13] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *Proceedings of SIGMOD*, 2000.

[14] W. Kießling. Foundations of preferences in database systems. In *Proceedings of VLDB*, 2002.

[15] W. Kießling, B. Hafenrichter, S. Fischer, and S. Holland. Preference XPATH: a query language for E-commerce. In *Konferenz f`ur Wirtschaftsinformatik* Aaugsburg, Germany, 2001.

[16] B. Hafenrichter and W. Kießing. Optimization of relational preference queries. *Australasian Database Conference (ADC)*, 2005.

[17] N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML Documents. In *Proceedings of SIGIR*, 2001.

[18] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML Data with relevance ranking. In *Proceedings of EDBT*, 2002.

[19] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of SIGMOD*, 2001.