On the Discovery of Conserved XML Query Patterns for Evolution-Conscious Caching

Sourav S. Bhowmick

School of Computer Engineering, Nanyang Technological University, Singapore
assourav@ntu.edu.sg

Abstract. Existing XML query pattern-based caching strategies focus on extracting the set of frequently issued *query pattern trees* (QPT) based on the support of the QPTs in the history. These approaches ignore the *evolutionary* features of the QPTs. In this paper, we propose a novel type of query pattern called *conserved query paths* (CQP) for efficient caching by integrating the *support* and *evolutionary* features together. CQPs are paths in QPTs that never change or do not change *significantly* most of the time (if not always) in terms of their support values during a specific time period. We proposed a set of algorithms to extract *frequent* CQP*s* (FCQPs) and *infrequent* CQP*s* (ICQPs) and rank these query paths using evolution-conscious *ranking functions*. Then, these ranked query paths are used in *evolution-conscious caching* strategy for efficient XML query processing. Finally, we report our experimental results to show that our strategy is superior to previous QPT-based caching approaches.

1 Introduction

In a XML data repository, a collection of XML queries may be issued by different users over a period of time. These queries can be represented as a collection of *query pattern trees* (QPTs) [12]. Given such a query collection, a *frequent* XML *query pattern* refers to a rooted QPT that is a subtree of at least *minsup* fraction of XML queries. Recently, several algorithms [11,12,13] have been proposed to mine these frequent patterns from the historical query log and cache the corresponding query results to reduce the response time for future queries that are the same or similar. These techniques are primarily designed for static collection of XML queries and cannot handle evolution of query workload efficiently. Consequently, several incremental algorithms [4,6] have been proposed to address the issue of efficiently maintaining the frequent query patterns.

Our initial investigation revealed that existing frequent query pattern-based caching strategies are solely based on the concept of frequency without taking into account the temporal features of the evolving query workload. Every occurrence of a query subtree contributes equally to the caching strategy regardless of *when* the query was issued. Consequently, this may not always be an effective approach in many real-life applications. For instance, consider the two queries, QPT_2 and QPT_4, in Figure 1. Assume that QPT_2 had been issued many times in the past but rarely in recent times whereas QPT_4 is only formulated frequently in recent times. Interestingly, QPT_2 may still remain as a frequent query over

X. Zhou et al. (Eds.): DASFAA 2009, LNCS 5463, pp. 527–542, 2009.

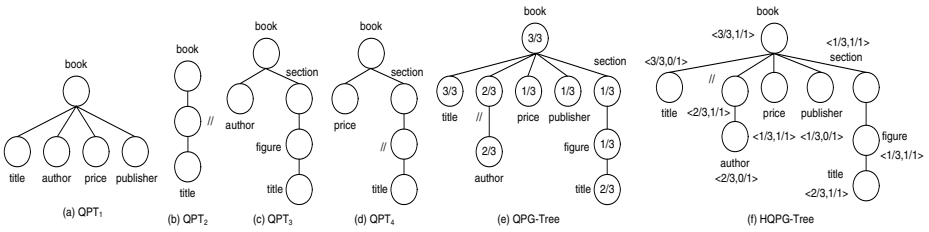


Fig. 1. QPTs, QPG-tree and HQPG-tree

the entire query collection due to its popularity in the past. On the other hand, in spite of its recent popularity, QPT_4 may be considered as *infrequent* with respect to the *entire* query collection in the history due to its lack of popularity in the past. Note that, it is indeed possible that more queries similar to QPT_4 are expected to be issued in the near future compared to queries similar to QPT_2.

In this paper, we propose a more effective and novel caching strategy that incorporates the evolutionary patterns of XML ***queries***. In our approach, each QPT consists of a set of *rooted query paths* (RQPs). Informally, a RQP in a QPT is a path starting from the root. For example, */book/section/figure* is a RQP of the XML query shown in Figure 1(c). In our approach, we first discover two groups of RQPs, the *frequent conserved query paths (*FCQP*)* and the *infrequent conserved query paths (*ICQP*)*, from the historical XML queries. Intuitively, *conserved query paths* (CQP) are RQPs whose *support* values never change or do not change significantly most of the times (if not always) during a time period. Here *support* represents the fraction of QPTs in the query collection that *includes* a specific RQP. *Hereafter, whenever we say changes to the* RQP*s/*QPT*s, we refer to the changes to the support of the* RQP*s.*

The second step of our approach is to build a more efficient evolution-conscious caching strategy using the discovered CQPs (FCQPs and ICQPs). Our strategy is based on the following principles. For RQPs that are FCQP, the corresponding query results should have higher priority to be cached since the support values of the RQPs is not expected to change significantly in the near future and the RQPs will be issued frequently in the future as well. Similarly, for RQPs that are ICQP, the corresponding query results should have lower caching priority.

We adopt a *path-level* caching strategy for XML queries instead of *twig-level* (subtree-level) caching. However, it does not hinder us in evaluating twig queries as such queries can be decomposed into query paths. In fact, decomposing twig queries into constituent paths has been used by several *holistic twig join* algorithms. Our focus in this paper is to explore how evolutionary characteristics of XML queries can enable us to design more efficient caching strategies. Our path-level, evolution-conscious caching approach can easily be extended to twig-level caching and we leave this as our future work. Importantly, we shall show later that our caching strategy can outperform a state-of-the-art twig-level, evolution-unaware caching approach [13].

Compared to existing caching strategies for XML data [5,2,12,13], our work differs as follows. Firstly, we use frequent and infrequent conserved RQPs instead of

frequent QPTs for caching strategies. Secondly, not only the frequency of the RQPs is considered, but also the evolution patterns of their support values are incorporated to make the caching strategy evolution-conscious. In summary, the main contributions of this paper are as follows. (a) We propose a set of metrics to measure the evolutionary features of QPTs (Section 2). (b) Based on the *evolution metrics*, two algorithms (D-CQP-MINER and R-CQP-MINER) are presented in Section 3 to discover novel patterns, namely *frequent* and *infrequent conserved query paths*. (c) A novel path-level evolution-conscious caching strategy is proposed in Section 4 that is based on the discovered CQPs. (d) Extensive experiments are conducted in Section 5 to show the efficiency and scalability of the CQP-MINER algorithms as well as effectiveness of our caching strategy.

2 Modeling Historical XML Queries

We begin by defining some terminology that we shall use later for representing historical XML queries. A *calendar schema* is a relational schema R with a constraint C, where $R = (f_n : D_n, f_{n-1} : D_{n-1}, \cdots, f_1 : D_1)$, f_i is the name for a calendar unit such as year, month, and day, D_i is a finite subset of positive integers for f_i, C is a Boolean-valued constraint on $D_n \times D_{n-1} \times \cdots \times D_1$ that specifies which combinations of the values in $D_n \times D_{n-1} \cdots D_1$ are valid. For example, suppose we have a calendar schema *(year:* {*2000, 2001, 2002*}*, month:* {*1, 2, 3,* $\cdots$ *,12*}*, day:* {*1, 2, 3,* $\cdots$ *, 31*}*)* with the constraint that evaluate $\langle y, m, d\rangle$ to be "true" only if the combination gives a valid date. Then, it is evident that $\langle$*2000, 2, 15*$\rangle$ is valid while $\langle$*2000, 2, 30*$\rangle$ is invalid. Hereafter, we use $*$ to represent any integer value that is valid based on the constraint.

Given a calendar schema R with the constraints C, a calendar pattern, denoted as $\mathbb{P}$, is a valid tuple on R of the form $\langle d_n, d_{n-1}, \cdots, d_1\rangle$ where $d_i \in D_i \cup \{*\}$. For example, given a calendar schema $\langle$*year, month, day*$\rangle$, the calendar pattern $\langle *,$ *1, 1*$\rangle$ refers to the time intervals "the first day of the first month of every year". Next we introduce the notion of *temporal containment*. Given a calendar pattern $\langle d_n, d_{n-1}, \cdots, d_1\rangle$ denoted as $\mathbb{P}_i$ with the corresponding calendar schema R, a timestamp t_j is represented as $\langle d'_n, d'_{n-1}, \cdots, d'_1\rangle$ according to R. The timestamp t_j is *contained* in $\mathbb{P}_i$, denoted as $t_j \simeq \mathbb{P}_i$, if and only if $\forall$ $1 \le l \le n$, $d'_l \in d_l$.

2.1 Representation of an XML Query

We adopt the *query pattern trees* (QPTs) [12,13] representation method in this paper. A *query pattern tree* is a rooted tree $QPT = \langle V, E\rangle$, where V is a set of vertex and E is the edge set. The root of the tree is denoted by $root(QPT)$. Each edge $e = (v_1, v_2)$ indicates node v_1 is the parent of node v_2. Each vertex v's label, denoted as $v.label$, is a tag value such that $v.label$ is in {"//", "*"} $\cup$ $tagSet$, where $tagSet$ is the set of all element and attribute names in the schema. Furthermore, if $v \in V$ and $v.label \in$ {"//", "$*$"} then there must be a $v' \in V$ such that $v' \in tagSet$ and is a child of v if $v.label =$ "//".

A QPT is a tree structure that represents the hierarchy structure of the predicates, result elements, and attributes in the XML query. Based on the definition of QPT, in existing approaches the *rooted subtree* of a QPT is defined to capture the common subtrees in a collection of XML queries [11,13]. However, in this paper, we are interested in *rooted query paths*, which can provide a finer granularity for caching than *rooted subtrees*. Rooted query paths are special cases of rooted subtrees. Given a QPT $QPT = \langle V, E \rangle$, $RQP = \langle V', E' \rangle$ is a *rooted query path* of QPT, denoted as $RQP \subseteq QPT$, such that (1) $Root(QPT) = Root(RQP)$ and (2) $V' \subseteq V$, $E' \subseteq E$, and RQP is a path in QPT. For example, */book/section/figure* is a RQP in Figure 1(c).

2.2 Representation of XML Query History

Each QPT is represented as a pair (QPT_i, t_i), where t_i is the timestamp recording the time when QPT_i was issued. As a result, the collection of queries (QPTs) can be represented as a sequence $\langle (QPT_1, t_1), (QPT_2, t_2), \cdots, (QPT_n, t_n) \rangle$, where $t_1 \leq t_2 \leq \ldots \leq t_n$. Then, a *Query Pattern Group* (QPG) is a bag of QPTs $[(QPT_i, t_i), (QPT_{i+k}, t_{i+k}), \cdots, (QPT_j, t_j)]$ such that $1 \leq (i, j) \leq n$ and $\forall\ m\ (i \leq m \leq j)$, $t_m \simeq \mathbb{P}_x$ where $\mathbb{P}_x$ is the user-defined calendar pattern. Observe that the QPTs in a specific QPG are issued within the same calendar pattern according to the calendar schema. Users can define their own time granularity according to the workload and application-specific requirements.

The sequence of QPTs can now be partitioned into a sequence of query pattern groups denoted as $\langle QPG_1, QPG_2, \cdots, QPG_k \rangle$. The occurrences of all QPTs in a QPG are considered to be *equally* important. In our approach, we compactly represent each QPG as a *query pattern group tree (*QPG*-tree)*.

Definition 1. Query Pattern Group Tree (QPG-tree): *Let $QPG = [QPT_i, QPT_{i+1}, \cdots, QPT_j]$ be a query pattern group. A query pattern group tree is a 3-tuple tree, denoted as $T_G = \langle V, E, \aleph \rangle$, where V is the vertex set, E is the edge set, and $\aleph$ is a function that maps each vertex to the support value of the corresponding rooted query path (*RQP*), such that $\forall\ RQP \subseteq QPT_k$, $i \leq k \leq j$, there exists a rooted query path, $RQP' \subseteq T_G$, that is extended included to RQP.*

Consider the three QPTs in Figures 1(a), (b), and (c). The corresponding QPG-tree is shown in Figure 1(e). The QPG-tree includes all RQPs and records the *support* values (the values inside the nodes of the RQPs in the figure). Given a query pattern group QPG_i, the *support* of a RQP in QPG_i is defined as $\Phi_i(\text{RQP}) = K / L$, where K denotes the number of times the RQP is *extended included* in the QPTs in QPG_i and L denotes the number of QPTs in QPG_i. When the RQP is obvious from the context, the support is denoted as Φ_i. Note that the traditional notion of *subtree inclusion* [9] is too restrictive for QPTs where handling of wildcards and relative paths are necessary. Hence, the concept of *extended subtree inclusion*, a sound approach to testing containment of query pattern trees, was proposed by Yang et al. [11] to count the occurrence of a tree pattern in the database. Here, we adopt this concept in the context of RQPs. Given two rooted query paths, RQP_1 and RQP_2, $RQP_1 \prec RQP_2$ denotes that RQP_1 is

extended included in RQP_2. Our definition of extended inclusion is similar to that of Yang et al. [11]. The only difference is that we assume the subtrees are RQPs. The formal definition is given in [15].

Since there can be a sequence of QPGs in the history, the mean support value of a RQP is represented as *Group Support Mean* (GSM). That is, let $\langle QPG_1, QPG_2, \cdots, QPG_n\rangle$ be a sequence of QPGs in the history. The GSM of a rooted query path, $RQP \subseteq QPG_i$ $(0 \leq i \leq n)$, denoted as $\overline{\Phi}(RQP)$, is defined as $\frac{1}{n}\sum_{i=1}^{n}\Phi_i$.

To facilitate discovery of specific patterns from the evolution history of the RQPs in the QPG-trees, we propose to merge the sequence of QPG-trees into a "global" tree called *historical* QPG-*tree* (HQPG-*tree*). It is similar to the idea of QPG-tree except for the function $\aleph$. In QPG-tree, the $\aleph$ function is used to map each vertex to a single support value of the rooted path at that vertex. In the definition of HQPG-tree, $\aleph$ is replaced by Ψ function which is used to map each vertex to a *sequence* of support values. For example, Figure 1(f) shows an example HQPG-tree by partitioning the QPTs in Figures 1(a), (b), (c), and (d) into two QPGs. The first three QPTs are in one group, while the last is in another group. The sequence of values associated with each vertex in Figure 1(f) corresponds to the support values. The formal definition is given in [15].

2.3 Evolution Metrics

Given a sequence of historical support values of a RQP, we can undertake two approaches to measure its evolutionary characteristics. First, in the *regression-based* approach, the evolution metric computes the "degree" of evolution (or conservation) from the sequence directly. Second, in the *delta-based* approach, we first compute the changes to consecutive support values in the sequence and then quantify the evolution characteristics of the RQP using a set of *delta-based* evolution metrics.

Regression-based Evolution Metric: Intuitively, the evolutionary pattern of a RQP can be modeled using regression models [10]. We propose a metric called *query conservation rate* to monitor the changes to supports of query paths using the linear regression model:$\Phi_t(RQP) = \Phi_0(RQP) + \lambda t$, *where* $1 \leq t \leq n$. Here the idea is to find a "best-fit" straight line through a set of n data points $\{(\Phi_1(RQP), 1), (\Phi_2(RQP), 2), \cdots, (\Phi_n(RQP), n)\}$, where $\Phi_0(RQP)$ and λ are constants called *support intercept* and *support slope*, respectively. The most common method for fitting a regression line is the method of least-squares [10]. By applying the statistical treatment known as linear regression to the data points, the two constants, $\Phi_0(RQP)$ and λ, can be determined. The correlation coefficient, denoted as r, can then be used to evaluate how the regression fits the data points actually.

Definition 2. Query Conservation Rate: *Let* $\langle \Phi_1, \Phi_2, \cdots, \Phi_n\rangle$ *be the sequence of historical support values of the rooted query path* RQP. *The query conservation rate of* RQP *is defined as* $\mathbb{R}(RQP) = r^2 - |\lambda|$ *where* $\lambda = \frac{\sum_{i=1}^{n} i\Phi_i - \sum_{i=1}^{n}\Phi_i\sum_{i=1}^{n} i}{n\sum_{i=1}^{n} i^2 - (\sum_{i=1}^{n} i)^2}$ *and* $r = \frac{n\sum_{i=1}^{n}(\Phi_i * i) - (\sum_{i=1}^{n}\Phi_i)(\sum_{i=1}^{n} i)}{\sqrt{[n\sum_{i=1}^{n}(\Phi_i)^2 - (\sum_{i=1}^{n}\Phi_i)^2][n\sum_{i=1}^{n} i^2 - (\sum_{i=1}^{n} i)^2]}}$.

Note that the larger the absolute value of the support slope, the more significantly the support changes over time. At the same time, the larger the value of r^2, the more accurate is the regression model. Hence, the larger the query conservation rate $\mathbb{R}(RQP)$, the support values of the RQP change less significantly or are more *conserved*. Also it can be inferred that $0 \leq \mathbb{R}(RQP) \leq 1$.

Delta-based Evolution Metrics: We now define a set of evolution metrics that are defined based on the changes to the support values of a RQP in consecutive QPG pairs. We begin by defining the notion of *support delta*. Let QPG_i and QPG_{i+1} be any two consecutive QPGs. For any rooted query path, RQP, the *support delta* of RQP from ith QPG to $(i+1)th$ QPG, denoted as $\delta_i(RQP)$, is defined as $\delta_i(RQP) = |\Phi_{i+1}(RQP) - \Phi_i(RQP)|$.

The *support delta* measures the changes to support of a RQP between any two consecutive QPGs. Obviously, a low δ_i is important for a RQP to be conserved. Hence, we define the *support conservation factor* metric to measure the percentage of QPGs where the support of a specific RQP changes *significantly* from the preceding QPG.

Definition 3. Support Conservation Factor: *Let* $\langle\ QPG_1, \cdots, QPG_n\ \rangle$ *be a sequence of* QPG*s . For any rooted query path, RQP, the support conservation factor in this sequence, denoted as* $\mathbb{S}(\alpha, RQP)$*, where* α *is the user-defined threshold for support delta, is defined as* $\mathbb{S}(\alpha, RQP) = \frac{\sum_{i=1}^{n-1} d_i}{n-1}$ *where (a) if* $\delta_i(RQP) \geq \alpha$ *then* $d_i = 1$*; (b) if* $\delta_i(RQP) < \alpha$ *then* $d_i = 0$.

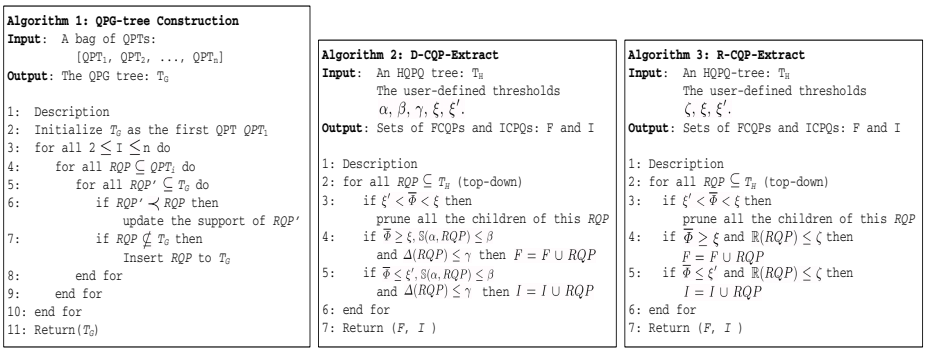
Observe that the smaller the value of $\mathbb{S}(\alpha, RQP)$ is, the less significant is the change to the support values of the RQP. Consequently, at first glance, it may seem that a low $\mathbb{S}(\alpha, RQP)$ implies that the RQP is conserved. However, this may not be always true as small changes to the support values in the history may have significant effect on the evolutionary behavior of a RQP over time. We define the *aggregated support delta* metric to address this.

Definition 4. Aggregated Support Delta: *Let* $\langle QPG_1, QPG_2, \cdots, QPG_n \rangle$ *be a sequence of* QPG*s in the history. The aggregated support delta of RQP, denoted as* $\Delta(RQP)$*, is defined as:* $\Delta(RQP) = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n-1} (\Phi_i - \Phi_{i+1})^2}$.

3 CQP-Miner Algorithms

We begin by formally presenting two definitions for CQPs by using the regression-based metric and delta-based metrics, respectively.

Definition 5. Conserved Query Path (CQP): *A RQP is a conserved query path in a sequence of* QPG*s iff any one of the following conditions is true: (a)* $\mathbb{R}(RQP) \leq \zeta$ *where* ζ *is the threshold for query conservation rate; (b)* $\mathbb{S}(\alpha, RQP) \leq \beta$ *and* $\Delta(RQP) \leq \gamma$ *where* α*,* β*, and* γ *are the thresholds for support delta, support conservation factor, and aggregated support delta, respectively.*



(a) QPG-tree Contruction Algorithm (b) D-CQP-Extract Algorithm (c) R-CQP-Exxtract Algorithm

Fig. 2. Algorithms for CQP mining

There are two variants of CQPs, *frequent conserved query paths* (FCQPs) and *infrequent conserved query paths*(ICQPs), which are important for our caching strategy. Both of them have the following characteristics: (a) the support values of the RQPs are either large enough or small enough; and (b) their support values do not evolve significantly in the history.

Definition 6. FCQP and ICQP: *Let* RQP *be a conserved query path. Let* ξ *and* ξ' *be the minimum and maximum group support mean (*GSM*) thresholds, respectively. Also,* $\xi > \xi'$*. Then, (a)* RQP *is a Frequent Conserved Query Path (*FCQP*) iff* $\overline{\Phi}(RQP) \geq \xi$*; (b)* RQP *is an Infrequent Conserved Query Path (*ICQP*) iff* $\overline{\Phi}(RQP) \leq \xi'$*.*

3.1 Mining Algorithms

Given a collection of historical XML queries, the objective of conserved query paths mining problem is to extract the frequent and infrequent CQPs. Using the delta-based and regression-based evolution metrics, we present two algorithms to extract the sets of FCQPs and ICQPs. We refer to these algorithms as D-CQP-Miner and R-CQP-Miner, respectively. Each algorithm consists of the following two major phases.

HQPG-tree Construction Phase: Given a collection of XML queries, an HQPG-tree is constructed in the following way. Firstly, the queries are transformed into QPTs. Then, the QPTs are partitioned into groups based on the timestamps and user-defined calendar pattern, where each QPG is represented as a QPG-tree. Next, the sequence of QPG-trees are merged together into an HQPG-tree. We elaborate on the construction of the QPG-tree and merging QPG-trees. The algorithm of constructing the QPG-tree is shown in Figure 2(a).

The algorithm of merging the sequence of QPG-trees into the HQPG-tree is similar to the above algorithm. The only difference is that rather than increasing the support values of the corresponding RQPs, a vector that represents the historical support values is created for each RQP. If the RQP does not exist in the

HQPG-tree, then the vector of supports for this RQP should be a vector starting with *i-1* number of *0*s, where i is the ID of the current query pattern group.

CQP Extraction Phase: Given the HQPG-tree, the FCQPs and ICQPs are extracted based on the user-defined thresholds for the corresponding evolution metric(s). Corresponding to the two definitions of CQPs, two algorithms are presented. The first algorithm is based on the delta-based evolution metrics and the second one is based on the regression-based evolution metric. We refer to these two algorithms as D-CQP-*Extract* and R-CQP-*Extract*, respectively. In both algorithms, the top-down traversal strategy is used to enumerate all candidates of both frequent and infrequent CQPs. We use the top-down traversal strategy based on the downward closure property of the GSM values for RQPs.

Lemma 1. *Let RQP_1 and RQP_2 be two rooted query paths in an* HQPG*-tree. If RQP_1 is included in RQP_2, then $\overline{\Phi}(RQP_1) \geq \overline{\Phi}(RQP_2)$.*

Due to space constraints, the proof is given in [15]. Based on the above lemma, we can prune the HQPG-tree during the top-down traversal. That is, for RQPs whose $\overline{\Phi}$ are smaller than ξ, no extensions of the RQPs can be FCQPs. Similarly, for RQPs whose $\overline{\Phi}$ are smaller than ξ', all of their extensions also satisfy this condition to be ICQPs. The D-CQP-*Extract* algorithm is shown in Figure 2(b). We first compare the values of $\overline{\Phi}$ with the thresholds of GSM. In this case, some candidates can be pruned. After that, the value of $\mathbb{S}(\alpha, RQP)$ is calculated and compared with β. Note that as $\mathbb{S}(RQP)$ is expensive to compute, it is only calculated for the candidates that satisfy all other constraints. The R-CQP-*Extract* algorithm (Figure 2(c)) is similar to the D-CQP-*Extract*, the only difference being the usage of different metrics.

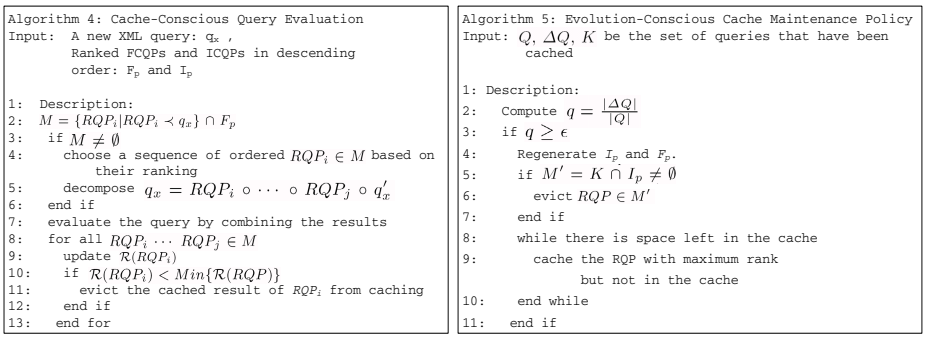
4 Evolution-Conscious Caching

We now present how to utilize the discovered CQPs to build the evolution-conscious cache strategy. There are two major phases, the CQP *ranking phase* and the *evolution-conscious caching (ECC) strategy phase*.

4.1 The CQP Ranking Phase

In this phase, we rank the CQPs discovered by the CQP-MINER algorithm using a *ranking function*. The intuitive idea is to assign high rank scores to query paths that are expected to be issued frequently. Note that there are other factors such as the query evaluation cost and the query result size that are important for designing effective caching strategy [11,12].

Definition 7. Ranking Functions: *Let the cost of evaluating a* RQP *(denoted as $Cost_{eval}(RQP)$) is the time to execute this query against the* XML *data source without any caching strategy, while the size of the result (denoted as $|result(RQP)|$) is the actual size of the view that stores the result. Then the ranking function, $\mathcal{R}$, is defined as: (a) If* D-CQP-MINER *is used to extract* ICQP*s and*



```
Algorithm 4: Cache-Conscious Query Evaluation
Input:  A new XML query: q_x ,
        Ranked FCQPs and ICQPs in descending
        order: F_p and I_p

1:  Description:
2:  M = {RQP_i | RQP_i ≺ q_x} ∩ F_p
3:   if M ≠ ∅
4:     choose a sequence of ordered RQP_i ∈ M based on
           their ranking
5:     decompose q_x = RQP_i ∘ ··· ∘ RQP_j ∘ q'_x
6:   end if
7:   evaluate the query by combining the results
8:   for all RQP_i ··· RQP_j ∈ M
9:     update R(RQP_i)
10:    if R(RQP_i) < Min{R(RQP)}
11:      evict the cached result of RQP_i from caching
12:    end if
13:  end for
```

(a) Cache-Conscious Query Evaluation Algorithm

```
Algorithm 5: Evolution-Conscious Cache Maintenance Policy
Input: Q, ΔQ, K be the set of queries that have been
       cached

1: Description:
2:   Compute q = |ΔQ| / |Q|
3:   if q ≥ ε
4:     Regenerate I_p and F_p.
5:     if M' = K ∩ I_p ≠ ∅
6:       evict RQP ∈ M'
7:     end if
8:     while there is space left in the cache
9:       cache the RQP with maximum rank
               but not in the cache
10:    end while
11:  end if
```

(b) Evolution-Conscious Cache Maintenance Policy Algorithm

Fig. 3. Algorithms for evolution-conscious caching

FCQPs, *then* $\mathcal{R}(RQP) = \frac{Cost_{eval}(RQP) \times \overline{\Phi}(RQP)}{\mathbb{S}(\alpha, RQP) \times \Delta(RQP) \times |result(RQP)|}$; *(b) If* R-CQP-MINER *is used to extract* ICQPs *and* FCQPs, *then* $\mathcal{R}(RQP) = \frac{Cost_{eval}(RQP) \times \overline{\Phi}(RQP)}{\mathbb{R}(RQP) \times |result(RQP)|}$.

Observe that we have two variants of the ranking function as our ranking strategy depends on the two sets of evolution metrics used in the regression-based (R-CQP-MINER) and delta-based (D-CQP-MINER) CQPs discovery approaches. Particularly, these evolution metrics are used to estimate the expected number of occurrences of the query paths. The remaining factors are used in the similar way as they are used in other cache strategies [3,8,12].

4.2 The ECC Strategy Phase

The goal of this phase is to construct an evolution-conscious caching strategy that utilizes the ranked FCQPs and ICQPs in such a way that the query processing cost for future incoming queries is minimized. As the cache space is limited, the basic strategy is to cache the results for the FCQPs with the *largest* rank scores by replacing the cached results of the RQPs with *smaller* rank scores.

We first introduce the notion of *composing query*. Suppose at time t_1, the cache contains a set of views $V = \{V_1, V_2, \cdots, V_n\}$ and the corresponding queries are $Q = \{Q_1, Q_2, \cdots, Q_n\}$. When a new query Q_{n+1} comes, it inspects each view V_i in V and determines whether it is possible to answer Q_{n+1} from V_i. View V_i answers query Q_{n+1} if there exists another query C which, when executed on the result of Q_i, gives the result of Q_{n+1}. It is denoted by $C \circ Q_i = Q_{n+1}$, where C is called the *composing query* (CQ). When a view answers the new query, we have a *hit*, otherwise we have a *miss*.

Cache-conscious query evaluation: Figure 3(a) describes the query evaluation strategy. When a new query q_x appears, it may match to more than one of the RQPs in the set of FCQPs (which are denoted as M). Hence, q_x can be considered to be the join of many $RQPs$ and the composing query q'_x. Formally, $q_x = RQP_1 \circ RQP_2 \cdots, RQP_j \circ q'_x$, where $RQP_1, RQP_2, \cdots, RQP_j$ are

the cached RQPs with the highest rank scores and are contained in q_x, q'_x is the composing query that does not contain any of the RQPs in the cache. The answers are obtained by evaluating the composing query and joining the corresponding results (Lines 2-7). Next, for all $RQPs$ that are contained in M, the corresponding ranks are updated with respect to the changes of $\overline{\Phi}$ (Lines 8-9). If the rank for any of these $RQPs$ falls below the minimum value of these RQPs in the cache, then the corresponding query results will be evicted (Lines 10-12). Note that we do not update the values of evolution metrics of ICQPs and FCQPs during the caching process. Rather, it is done off-line as discussed below.

Evolution-conscious cache maintenance policy: One can observe that under heavy query workload, mining FCQPs and ICQPs frequently during evaluation of every new query can be impractical. Hence, rather than computing new sets of FCQPs and ICQPs whenever a new query appears, we recompute these CQPs *only when the number of new queries that have been issued, in comparison with the set of historical queries, is larger than some factor q*. Note that this mining process can be performed off-line.

Formally, let t_p be the most recent time when we computed the sets of FCQPs and ICQPs in the history. Let $|Q|$ denote the number of XML queries in the collection at t_p. Assume that we recompute the sets of FCQPs and ICQPs at time t_n where $t_n > t_p$. Let $|\Delta Q|$ be the set of new queries that are added during t_p and t_n. Then, $q = \frac{|\Delta Q|}{|Q|}$.

The algorithm for query evaluation is shown in Figure 3(b). First, it computes the q value. If q is greater than or equal to some threshold ϵ then the FCQPs and ICQPs are updated off-line. In Section 5.2, we shall empirically show that $\epsilon = 0.5$ produces good results. If the RQPs that have been cached are in the list of regenerated ICQPs, then the corresponding results in the cache have to be evicted (Lines 5-7). Consequently, there may be some space in the cache available that can be utilized. If the space is enough, then cache those RQPs in F_p having maximum rank but have not been cached yet (Lines 8-10).

5 Performance Evaluation

The mining algorithms and the caching strategy are implemented in Java. All the experiments were conducted on a Pentium IV PC with a 1.7Ghz CPU and 512MB RAM, running MS Windows 2000. We use two set of synthetic datasets generated based on the DBLP.DTD (http://dblp.uni-trier.de/xml/dblp.dtd) and SSPLAY.DTD (http://www.kelschindexing.com/shakesDTD.html). Firstly, a DTD graph is converted into a DTD tree by introducing some "//" and "*" nodes. Then, all possible rooted query paths are enumerated. Similar to [6,12,13], the collection of QPTs is generated based on the set of RQPs using the Zipfian distribution and these QPTs are randomly distributed in the temporal dimension. Example of two sets of QPTs in the DBLP and SSPLAY datasets is given in [15]. Each basic dataset consists of up to 3,000,000 QPTs, which are divided into 1000 QPGs. The characteristics of the datasets are shown in Figure 4(a).

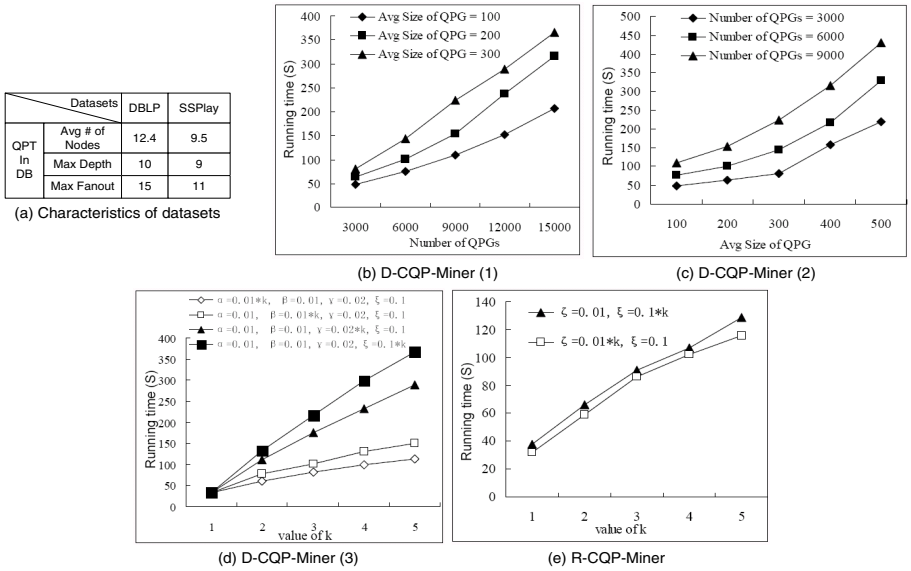


Fig. 4. Datasets and performance of CQP-Miner

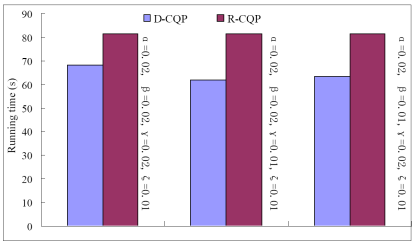
5.1 CQP-Miner

Algorithm Efficiency: We evaluate the efficiency by varying the average size of QPGs and the number of QPGs (the size of the time window). Figures 4(b) and (c) show the running time of the D-CQP-Miner when the size of the dataset increases. In the first case, the number of QPGs is increased while the average size of each QPG is fixed. In the second case, the average size of each QPG increases while the number of QPGs is fixed. The DBLP dataset is used and the parameters are fixed as follows: $\alpha = 0.02$, $\beta = 0.05$, $\gamma = 0.02$, and $\xi = 0.25$. Also, we set $\xi' = \xi/10$. It can be observed that when the size of the dataset increases, the running time increases as well. The reason is intuitive as the size of the HQPG-tree becomes larger, it requires more time for the tree construction and handling large number of candidate CQPs. The running time of the R-CQP-Miner shows a similar trend. Due to space constraints, the reader may refer to [15] for details.

Effects of Thresholds: As there are four thresholds: α, β, γ, and ζ for the D-CQP-Miner, experiments are conducted by varying one of the them and fixing the others. For instance, in Figure 4(d), "$\alpha = 0.01 * k$, β=0.01, γ=0.02, ξ=0.1" means that we fix the values of β, γ and ξ, and vary α from 0.01 to 0.05 by varying k from 1 to 5. In this experiment, the DBLP dataset with 300,000 queries is used. The results in Figure 4(d) show that the running time of D-CQP-Miner increases with the threshold values. Observed that the changes to ξ and α have more significant effect on the running time than the changes to β and γ. This is because ξ affect the total number of FCQPs and ICQPs and the values of α affect both support deltas and support conservation factors.

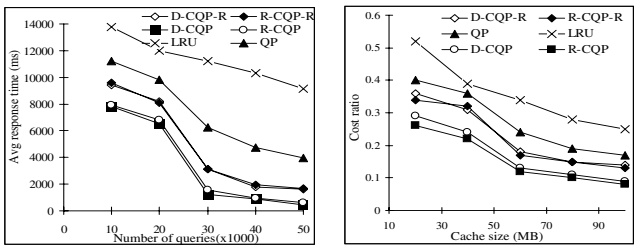
α	β	γ	ζ	overlap	o@10	o@30	o@60
0.01	0.01	0.01	0.01	0.66	1	0.83	0.77
0.01	0.01	0.02	0.01	0.71	1	0.87	0.83
0.01	0.02	0.01	0.01	0.85	1	1	0.88
0.02	0.01	0.01	0.01	0.79	1	0.93	0.79
0.02	**0.02**	**0.02**	**0.01**	**0.93**	**1**	**1**	**1**
0.02	**0.02**	**0.01**	**0.01**	**0.97**	**1**	**1**	**1**
0.02	**0.01**	**0.02**	**0.01**	**0.94**	**1**	**0.98**	**1**
0.01	0.02	0.02	0.01	0.91	1	1	0.91
0.03	0.02	0.01	0.01	0.88	1	1	0.89

(a) Mining Results



(b) Running Times

Fig. 5. Comparison of mining algorithms



(a) Avg response time (b) Cost ratio

Fig. 6. Performance of caching strategies

Similarly, the thresholds, ζ and ξ, are varied to evaluate their effects on the running time of the R-CQP-Miner. The results are shown in Figure 4(e). The SSPlay dataset with 900,000 queries is used. It can be observed that the running time increases with the thresholds. The reason is that when the values for any of the two parameters increase, the number of CQPs increases.

Comparison of Mining Results: As the two algorithms use different evolution metrics, to compare the mining results, we define the notion of *overlap* metric. Let F_D and I_D be the sets of FCQPs and ICQPs, respectively, in the D-CQP-Miner mining results. Let F_R and I_R be the sets of FCQPs and ICQPs in the R-CQP-Miner mining results. The *overlap* between the two sets of mining results is defined as: $Overlap = \frac{1}{2} \times (\frac{|F_D \cap F_R|}{|F_D \cup F_R|} + \frac{|I_D \cap I_R|}{|I_D \cup I_R|})$.

Basically, the *overlap* value is defined as the number of shared CQPs divided by the total number of unique CQPs in both mining results. Based on this definition, it is evident that the larger the *overlap* value, the more similar the mining results are. In this definition all the CQPs in the mining results are taken into consideration. However, in caching, only the top-k frequent/infrequent CQPs in the results are important. Hence, we define the notion of *overlap*@k metric. Let $C_D(k)$ and $C_R(k)$ be the sets of top-k conserved query paths in the D-CQP-Miner and R-CQP-Miner results, respectively, where $C_D(k) \subseteq F_D \cup I_D$ and $C_R(k) \subseteq F_R \cup I_R$. The *overlap*@k (denoted as $o@k$) is defined as: $o@k = \frac{|C_D(k) \cap C_R(k)|}{|C_D(k) \cup C_R(k)|}$.

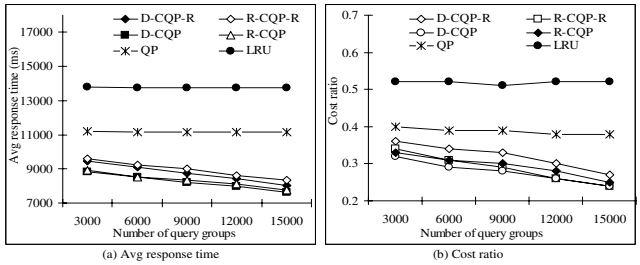


Fig. 7. Effect of QPG size

The experimental results with the SSPLAY dataset is shown in Figure 5(a). We vary the thresholds of the evolution metrics and compute the *overall* and *overall@k*. Interestingly, the *overlap* value can be very close to *1* when the threshold values are appropriately set. This indicates that both algorithms share a large number of CQPs even though they are based on different evolution metrics. Moreover, it can be observed that the top-*10* CQPs are exactly the same. Even for the top-*60* CQPs, the two categories of evolution metrics can produce identical sets of CQPs under appropriate threshold values. This is indeed encouraging as it indicates that both the regression-based and delta-based evolution metrics can effectively identify the top-k CQPs that are important for our caching strategy.

Comparison of Running Times: We now compare the running times of the two algorithms when they produce identical top-k CQPs under appropriate thresholds. We choose the three sets of threshold values shown in Figure 5(a) that can produce identical top-60 CQPs (shaded region in the table). Figure 5(b) shows the comparison of the running time. The DBLP dataset is used and ξ is set to *0.1*. It can be observed that D-CQP-MINER is faster than R-CQP-MINER when they produce the same top-60 CQPs.

5.2 Evolution-Conscious Caching

We have implemented the caching strategy by modifying the replacement policies of LRU with the knowledge of FCQPs and ICQPs as stated in the previous section. From the original collections of QPTs, some QPTs are chosen as the basic query paths and are extended to form the future queries. To select the basic query paths, queries that are issued more recently have a higher possibility of being chosen. That is, given a sequence of n QPGs, $\frac{n-i}{2^{i+1-n}}$ QPTs are selected from the ith group. Then, the set of selected queries are extended according to the corresponding DTD. The future queries are generated by extending the previous query paths with the randomly selected query paths. Note that for each of the following experiments, *10* sets of queries are generated for evaluation and the figures show the average performance. The QPTs used for generating examples of the *10* sets of queries are given in [15].

We use the same storage scheme as in [12]. That is, we use the index scheme of [7] to populate the SQL Server 2000 database and create the corresponding

indexes. The system accepts tree-patterns as its queries, and utilizes structural join method [1] to produce the result. No optimization techniques are used.

Basically, six caching strategies are implemented: the D-CQP-MINER and R-CQP-MINER-based strategies (denoted as DCQP and RCQP, respectively), D-CQP-MINER and R-CQP-MINER-based strategies without a ranking function (denoted as DCQP-R and RCQP-R, respectively), the original LRU-based caching strategy (denoted as LRU), and the state-of-the-art frequent query pattern-based caching strategy (2PX-MINER [13] based caching strategy denoted as QP). Note that the FCQPs and ICQPs used in the following experiments are discovered using the D-CQP-MINER and R-CQP-MINER, by setting $\alpha = 0.02$, β=0.02, γ=0.01, ζ=0.01, and $\xi = 0.2$.

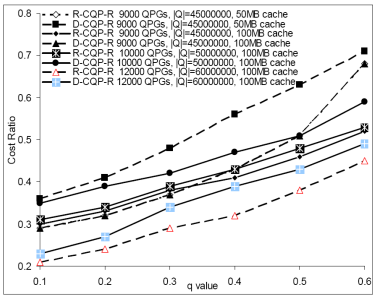
Average Response Time: The *average response time* is the average time taken to answer a query. It is defined as the ratio of total response time for answering a set of queries to the total number of queries in this set. Note that the query response time includes the time for ranking the CQPs (The CQP ranking phase). Figure 6(a) shows the average response time of the six approaches while varying the number of queries from 10,000 to 50,000 with the cache size fixed at 40MB. We make the following observations. First, as the number of queries increases, the average response time decreases. This is because when the number of queries increases, more historical behaviors can be incorporated and the frequent query patterns and conserved query paths can be more accurate. Second, DCQP, DCQP-R, RCQP, and RCQP-R perform better than QP and LRU. Particularly, when the number of queries increases, the gaps between our approaches and the existing approaches increases as well. For instance, our caching strategies can be up to 5 times faster than the QP approach and 10 times faster than the LRU approach when the number of queries is up to 50,000. Third, the rank-based evolution-conscious caching strategies outperform the rank-unconscious caching strategies highlighting the benefits of using the ranking functions.

Cost ratio: The *cost ratio* represents the query response time using different types of caching strategies against the response time without any caching strategy for all query examples. Figure 6(b) shows the performance of the six caching strategies in terms of the cost ratio measure. The number of queries is fixed at 2000, while the cache size varies from 20MB to 100MB (for the SSPLAY dataset). It can be observed that DCQP, DCQP-R, RCQP, and RCQP-R perform better than QP and LRU. Particularly, observe that the ratio difference between state-of-the-art QP approach and LRU is between $0.09 \sim 0.12$. If we consider this as the benchmark then observed further difference of $0.1 \sim 0.13$ between our approach and QP is significant. In other words, the idea of including evolutionary feature of queries for caching is an effective strategy.

Number of QPGs: Figures 7(a) and (b) show how the average response time and cost ratio change when the number of QPGs increases. The SSPLAY dataset is used and the average size of each QPG is 300. We vary the number of QPGs from 3,000 to 15,000. Observe that the evolution-conscious caching strategies perform better when there are more QPGs. This is because when the number of QPGs is large, our CQPs are more accurate.

q	LRU	R-CQP-R	D-CQP-R	QP
0.1	0.52	0.29	0.36	0.39
0.2	0.52	0.32	0.41	0.45
0.3	0.52	0.37	0.48	0.53
0.4	0.52	0.43	0.56	0.61
0.5	0.52	0.51	0.63	0.74
0.6	0.52	0.68	0.71	0.86

(a) Effects of q on cost ratio (1)



(b) Effects of q on cost ratio (2)

Fig. 8. Effects of q on cost ratio

Maintenance cost of ICQPs and FCQPs: As mention in Section 4.2, the sets of FCQPs and ICQPs need to be updated after certain number of queries are issued. In this experiment, we empirically determine the threshold value ϵ such that as long as $q < \epsilon$ we do not need to update the ICQPs and FCQPs. We first vary q to study its effect on the quality of our caching strategy. Note that from the running cost point of view, the larger the value of q, the lesser is the overhead. Figure 8(a) shows the performance of our proposed approaches compared to the LRU and QP approaches (in terms of cost ratio). We set $|Q| = 45000000$ (9000 QPGs) and the cache size is fixed to 50MB. It can be observed that the cost ratio increases with the increase in q for all approaches except the LRU-based approach. For the QP approach, rather than repeatedly updating the frequent query patterns whenever new queries are issued, the same strategy of periodically updating the mining results is used. It can be observed that the performance of our proposed RCQP-R and DCQP-R are better than the QP approach for any q value. Furthermore, RCQP-R and DCQP-R are better than the LRU approach in most cases when $q < 0.5$.

In Figure 8(b) we vary $|Q|$ and the cache size to study the effect of q on the cost ratio. It can be observed that our approaches produce good performance in most cases when $q < 0.5$ ($\epsilon = 0.5$). That is, our approach can improve the query performance without updating the FCQPs and ICQPs as long as $|\Delta Q| < \frac{|Q|}{2}$.

6 Conclusions and Future Work

In this paper, we proposed a novel type of XML query pattern named conserved query paths (CQPs) for efficient caching. To the best of our knowledge, this is the first approach that integrates evolutionary features of XML queries along with frequency of occurrences for building an efficient caching strategy. Conserved query paths are rooted query paths (RQPs) in QPTs that never change or do not change significantly most of the time in terms of their support values during a specific time period. Based on two evolution metrics, we presented two algorithms (D-CQP-Miner and R-CQP-Miner) that extract frequent and

infrequent CQPs from the historical collection of QPTs. These CQPs are ranked according to our proposed ranking function and used to build the evolution-conscious caching strategy. Experimental results showed that the proposed algorithms can be effectively used to build more efficient caching strategies compared to state-of-the-art caching strategies. In future, we wish to explore how calendar pattern selection can be automated. Also, we would like to extend our framework to provide a more sophisticated probabilistic ranking function. Finally, we plan to investigate strategies to automate the maintenance of ICQPs and FCQPs.

Acknowledgement. The author wishes to acknowledge and thank Dr Qiankun Zhao for implementing the ideas discussed in this paper.

References

1. Al-Khalifa, S., Jagadish, H.V., et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In: ICDE (2002)
2. Chen, L., Rundensteiner, E.A., Wang, S.: Xcache: A Semantic Caching System for XML Queries. In: SIGMOD, p. 618 (2002)
3. Chen, L., Wang, S., Rundensteiner, E.: Replacement Strategies for XQuery Caching Systems. Data Knowl. Eng. 49(2), 145–175 (2004)
4. Chen, Y., Yang, L., et al.: Incremental Mining of Frequent XML Query Patterns. In: Perner, P. (ed.) ICDM 2004. LNCS, vol. 3275. Springer, Heidelberg (2004)
5. Hristidis, V., Petropoulos, M.: Semantic Caching of XML Databases. In: WebDB (2002)
6. Li, G., Feng, J., et al.: Incremental Mining of Frequent Query Patterns from XML Queries for Caching. In: ICDM (2006)
7. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. In: VLDB (2001)
8. Mandhani, B., Suciu, D.: Query Caching and View Selection for XML Databases. In: VLDB (2005)
9. Ramesh, R., Ramakrishnan, L.V.: Nonlinear Pattern Matching in Trees. JACM 39(2), 295–316 (1992)
10. Weisberg, S.: Applied Linear Regression, 2nd edn. Wiley, Chichester (1985)
11. Yang, L., Lee, M., et al.: Mining Frequent Query Patterns from XML Queries. In: DASFAA (2003)
12. Yang, L., Lee, M., et al.: Efficient Mining of XML Query Patterns for Caching. In: VLDB (2003)
13. Yang, L., Lee, M., et al.: 2pxminer: An Efficient Two Pass Mining of Frequent XML Query patterns. In: SIGKDD (2004)
14. Zaki, M.J.: Efficiently Mining Frequent Trees in a Forest. In: SIGKDD, pp. 71–80 (2002)
15. Bhowmick, S.S.: cqp-Miner: Mining Conserved XML Query Patterns For Evolution-Conscious Caching. Technical Report, CAIS-12-2007 (2007), `http://www.cais.ntu.edu.sg/~assourav/TechReports/CQPMiner-TR.pdf`