CQP-Miner: Mining 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 query pattern-based caching approaches.

1 Introduction

Semantic/query caching has played a key role in improving query performance in databases. With the emergence of XML as a standard for data representation and exchange on the Web, several researchers have examined different cache strategies for XML query evaluation [4, 10, 16, 17]. One of the efforts in this direction is to discover the *frequent query 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 [16].

In the literature, several algorithms such as XQPMiner [15], FastXMiner [16], and 2PXMiner [17] have been proposed to mine frequent query patterns. Given an XML data repository X, let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of XML queries issued against X. Then, the queries in Q can be transformed into a transactional database, $D = \{QPT_1, QPT_2, QPT_3, \cdots, QPT_n\}$, where each transaction is a *query pattern tree* (QPT) that corresponds to an XML query. A *frequent query pattern* refers to a rooted tree that is a subtree of at least *minsup* number of XML queries in the query collection, where *minsup* is a user-defined minimum support. For example, Figure 1 shows a set of query pattern trees (QPT) representing *four* XML queries. If the *minsup* is *two*, then Figure 1(e) is a frequent query pattern since it is an *extended subtree* of QPT_1 and QPT_4 according to the *extended subtree inclusion* concept in [16, 17]. Note that these frequent query pattern mining techniques are primarily designed for static collection of XML queries. Thus, they cannot handle evolution of query workload efficiently, because they have to compute the frequencies of candidate query patterns from scratch in order to get most up-to-date frequent query patterns. Consequently, several incremental algorithms [5, 8] have been recently proposed to address the issue of efficiently maintaining the frequent query patterns.

While the above techniques have certainly been innovative and powerful, 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. However, this may not always be an effective approach in many real-life applications. For instance, reconsider 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 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. However, existing techniques fail to exploit such evolutionary features of the queries while designing caching strategies.

In this paper, we propose a more effective and novel caching strategy that incorporates the evolutionary patterns of XML ***queries***. Given an XML data repos-

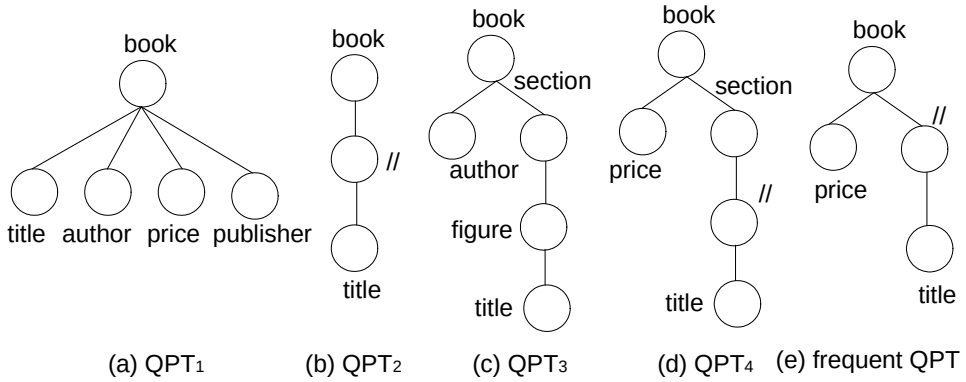


Figure 1: QPTs and frequent QPT.

itory, a collection of XML queries are issued by different users over a period of time. These queries can be represented as a collection of QPTs. 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. We define a set of *evolution metrics* to determine how *conserved* the RQPs are. *Hereafter, whenever we say changes to the* QPT*s*/RQP*s, we refer to the changes to the support values 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.

Observe that 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 widely used by several *holistic twig join* algorithms [2]. 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. More importantly, we shall show in Section 5 that our proposed caching strategy can outperform a state-of-the-art twig-level, evolution-unconscious caching approach [17].

In summary, the main contributions of this paper are as follows.

- We propose a set of metrics to measure the evolutionary features of QPTs (Section 2).
- 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*.
- A novel and efficient evolution-conscious caching strategy is proposed in Section 4 that is based on the discovered CQPs. 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.
- 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. Our study reveals that our caching strategy outperforms state-of-the-art approaches such as 2PX-MINER [17] and LRU-based caching approaches in terms of *average response time* and *cost ratio*.

2 Modeling Historical XML Queries

In this section, we propose a novel way of representing the historical collection of XML queries. Then, a set of *evolution metrics* is proposed to measure the evolutionary nature of the rooted query paths.

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. The reason to use the calendar schema is to exclude invalid time intervals that are generated due to the combinations of calendar units. Specifically, by modifying the constraint, users can further narrow down valid time intervals according to application-specific requirements. For instance, if we want to monitor the change patterns of XML queries on a daily basis, then a daily-based calendar schema can be defined and used. 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 $\forall\ 1 \leq l \leq n, d'_l \in d_l$. For example, given a calendar pattern $\langle *, 2, 12 \rangle$ with the calendar schema $\langle$ *week, day of the week, hour*$\rangle$, the timestamp *2008-05-24 12:28* is contained in this calendar pattern as it is not the second day of the week, while the timestamp *2008-05-27 12:08* is.

2.1 Representation of an XML Query

In existing XML query pattern mining approaches, XML queries are modeled as trees called *Query Pattern Trees* (QPTs) [16,17]. We adopt the QPT 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 $v.label \in \{\text{“}//\text{”}, \text{“}*\text{”}\} \cup tagSet\}$ where $tagSet$ is the set of all element and attribute names in the schema. Furthermore, if $v \in V$ and $v.label \in \{\text{“}//\text{”}, \text{“}*\text{”}\}$ then there must be a $v' \in V$ such that $v' \in tagSet$ and is a child of v if $v.label = \text{“}//\text{”}$.

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 [15, 17]. 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

In existing approaches, QPTs are modeled as transactions and the timestamps associated with QPTs are ignored. In our approach, 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

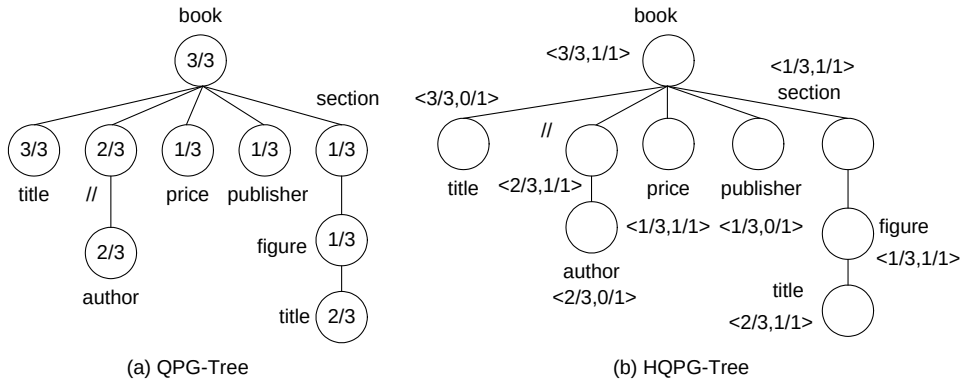


Figure 2: QPG-tree and HQPG-tree.

are considered to be *equally* important. In our approach, we compactly represent each QPG as a *query pattern group tree (*QPG*-tree)*. Formally, a QPG-tree is defined as follows.

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 2(a). 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* [11] 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. [15] 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. [15]. The only difference is that we assume the subtrees are RQPS. The formal definition is as follows.

Definition 2 Partial Ordering of Labels: *Given two labels y and y′, if $y = y'$, then we say $y \leq y'$. For any label $y \in tagSet$, we define $y \leq * \leq //$, that is, a node with label y matches a wildcard, which in turn matches a node with label '//'.*

Definition 3 Extended Inclusion of RQPs: *Let RQP_a and RQP_b be two* RQPs *with root nodes a and b, respectively. Let child(v) denotes the child node of v. Then we can recursively determine if RQP_a is included in RQP_b, denoted by $RQP_a \prec RQP_b$, as follows: $a \leq b$ and satisfies: (1) both a and b are a leaf nodes; or (2) a is a leaf node and b = '//', then $\exists$ child(b) such that $RQP_a \prec RQP_{child(b)}$; or (3) both a and b are non-leaf nodes, and one of the following holds:*

1. *For child(a), $\exists$ child(b) s.t $RQP_{child(a)} \prec RQP_{child(b)}$; or*
2. *b = '//' and $RQP_{child(a)} \prec RQP_b$; or*
3. *b = '//' and $\exists$ child(b) where $RQP_a \prec RQP_{child(b)}$.*

Note that as 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*).

Definition 4 Historical QPG-tree (HQPG-tree): *Let $\mathcal{G} = \langle T_{G_1}, T_{G_2}, \cdots, T_{G_i} \rangle$ be a sequence of* QPG*-trees. An* HQPG*-tree is a 3-tuple tree, denoted as $T_H = \langle V, E, \Psi \rangle$, where V is the vertex set, E is the edge set, and Ψ is a function that maps each vertex to a sequence of support values, such that $\forall$ $RQP \subseteq T_{G_k}$, $1 \leq k \leq i$, $\exists$ $RQP' \subseteq T_H$ that is identical to RQP.*

The idea of HQPG-tree is similar to the idea of QPG-tree except for the function Ψ. In the definition of QPG-tree, the $\aleph$ function is used to map each vertex to a single real value, which is the support value of the rooted path at that vertex; in the definition of HQPG-tree, the Ψ function is used to map each vertex to a sequence of support values. For example, Figure 2(b) 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 2 corresponds to the support values.

2.3 Evolution Metrics

In order to extract conserved query paths, it is important to define metric(s) that can quantify the evolutionary characteristics of a specific RQP in history. Intuitively, the lower the degree of evolution of a RQP with respect to its support values is, the more *conserved* it is in the history.

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. The justification for having two such approaches is that we wish to study the effect of different types of evolution metrics on the caching performance (discussed in Section 5).

Regression-based Evolution Metric: Intuitively, the evolutionary pattern of a RQP can be modeled using regression models [14]. 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 [14]. 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. Based on the above model we define the *query conservation rate* metric.

Definition 5 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*

$$\lambda = \frac{\sum_{i=1}^{n} i\Phi_i - \sum_{i=1}^{n} \Phi_i \sum_{i=1}^{n} i}{n\sum_{i=1}^{n} i^2 - (\sum_{i=1}^{n} i)^2}$$

$$r = \frac{n\sum_{i=1}^{n} (\Phi_i * i) - (\sum_{i=1}^{n} \Phi_i)(\sum_{i=1}^{n} i)}{\sqrt{[n\sum_{i=1}^{n} (\Phi_i)^2 - (\sum_{i=1}^{n} \Phi_i)^2][n\sum_{i=1}^{n} i^2 - (\sum_{i=1}^{n} i)^2]}}$$

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 rooted query path 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 6 Support Conservation Factor: *Let* $\langle\ QPG_1, \cdots, QPG_n\ \rangle$ *be a sequence of* QPGs *. 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*

$$\mathbb{S}(\alpha, RQP) = \frac{\sum_{i=1}^{n-1} d_i}{n-1} \text{ where } d_i = \begin{cases} 1, if\ \delta_i(RQP) \geq \alpha \\ 0, if\ \delta_i(RQP) < \alpha \end{cases}$$

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. For instance, suppose that the support value of a RQP keeps increasing (decreasing) with very small support delta (e.g., $\delta_i = 0.01$). If $\alpha = 0.02$ then it is possible for $\mathbb{S}(\alpha, RQP)$ to be low. When the number of QPGS is very large, it is possible that the support values of the RQP in the first few QPGS are significantly smaller (larger) than the support values of the RQP in the last few QPGS. In order to address this issue, we define the *aggregated support delta* metric.

Definition 7 Aggregated Support Delta: *Let* $\langle QPG_1, QPG_2, \cdots, QPG_n \rangle$ *be a sequence of* QPGs *in the history. The aggregated support delta of RQP, denoted as* $\Delta(RQP)$*, is defined as:* $\Delta(RQP) = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n-1} (\Phi_i - \Phi_{i+1})^2}$.

Observe that for two sequences of support values that have same mean and standard deviation values, the values of aggregated support delta can be substantially different. It can be observed that the value of Δ is within the range of [0, 1]. A larger value of Δ indicates that the support values of the corresponding RQP are not conserved over time.

3 CQP-Miner Algorithms

In this section, we present the algorithms for discovering *conserved query paths* (CQP). We begin by formally presenting two definitions for CQPS by using the regression-based metric and delta-based metrics, respectively.

Definition 8 Conserved Query Path (CQP): *A RQP is a conserved query path in a sequence of* QPGs *if and only if 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.*

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 9 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 two 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. As the process of transforming queries into QPTs and partitioning QPTs into groups are straightforward, we present the details of constructing the QPG-tree and merging QPG-trees.

The algorithm of constructing the QPG-tree is shown in Algorithm 1. Firstly, the QPG-tree is initialized as the first QPT in that group. Next, other QPTs in the group are compared with the QPG-tree to construct the structure of the QPG-tree as shown in Lines 3-10. Each RQP in the corresponding QPT is compared with the existing QPG-tree and the support values of the RQP′s in the QPG-trees that are extended included by this RQP are updated. Moreover, if this RQP is not included in the existing QPG-tree, then the QPG-tree is updated by inserting this RQP into the tree. This process iterates for all the RQPs in all the QPTs in the corresponding query pattern group. An example QPG-tree is shown in Figure 2(a) for the QPTs in Figures 1(a)-(c).

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. Figure 2(b) shows an example of HQPG-tree.

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

Algorithm 1: QPG-tree Construction

Input: A bag of QPTS: $[QPT_1, QPT_2, \ldots, QPT_n]$
Output: The QPG tree: T_G

Description
Initialize T_G as the first QPT QPT_1
for all $2 \leq i \leq n$ **do**
  for all $RQP \subseteq QPT_i$ **do**
    for all $RQP' \subseteq T_G$ **do**
      if $RQP' \prec RQP$ **then** update the support of RQP'
      if $RQP \notin T_G$ **then** Insert RQP to T_G
    end for
  end for
end for
Return(T_G)

Algorithm 2: D-CQP-Extract

Input: An HQPG tree: T_H
The user-defined thresholds $\alpha, \beta, \gamma, \xi, \xi'$.
Output: Sets of FCQPS and ICQPS: F and I

Description
for all $RQP \subseteq T_H$ (top-down)
  if $\underline{\xi'} < \overline{\Phi} < \xi$ **then** prune all the children of this RQP
  if $\overline{\Phi} \geq \xi$, $\mathbb{S}(\alpha, RQP) \leq \beta$ and $\Delta(RQP) \leq \gamma$ **then** $F = F \cup RQP$
  if $\overline{\Phi} \leq \xi'$, $\mathbb{S}(\alpha, RQP) \leq \beta$ and $\Delta(RQP) \leq \gamma$ **then** $I = I \cup RQP$
end for
Return (F, I)

Algorithm 3: R-CQP-Extract

Input: An HQPG-tree: T_H
The user-defined thresholds ζ, ξ, ξ'.
Output: Sets of FCQPS and ICQPS: F and I

Description
for all $RQP \subseteq T_H$ (top-down)
  if $\underline{\xi'} < \overline{\Phi} < \xi$ **then** prune all the children of this RQP
  if $\overline{\Phi} \geq \xi$ and $\mathbb{R}(RQP) \leq \zeta$ **then** $F = F \cup RQP$
  if $\overline{\Phi} \leq \xi'$ and $\mathbb{R}(RQP) \leq \zeta$ **then** $I = I \cup RQP$
end for
Return (F, I)

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 $\overline{\Phi}(RQP_1) \geq \overline{\Phi}(RQP_2)$.*

Proof 1 (SKETCH) Based on the definition of QPG-tree, it can be inferred that $\Phi_i(RQP_1) \geq \Phi_i(RQP_2)$ for all i $(1 \leq i \leq n)$ because whenever RQP_2 is extended included in a QPT, RQP_1 is also extended included in that QPT provided that RQP_1 is included in RQP_2. As a result, it is evident that $\overline{\Phi}(RQP_1) \geq \overline{\Phi}(RQP_2)$.

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 Algorithm 2. The idea is to 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. If $\mathbb{S}(\alpha, RQP) \leq \beta$, then the value of $\Delta(RQP)$ is calculated and compared with γ. Otherwise, we do not need to calculate the value of $\Delta(RQP)$. If $\mathbb{S}(RQP) \leq \gamma$, then the RQP is inserted into the corresponding group of FCQPs or ICQPs. 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 (Algorithm 3) 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 [15, 16].

Definition 10 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:*

- *If* D-CQP-MINER *is used to extract* ICQPs *and* FCQPs*, then*
$$\mathcal{R}(RQP) = \frac{Cost_{eval}(RQP) \times \overline{\Phi}(RQP)}{\mathbb{S}(\alpha, RQP) \times \Delta(RQP) \times |result(RQP)|}$$
- *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 [4, 10, 16].

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

Description:
$M = \{RQP_i | RQP_i \prec q_x\} \cap F_p$
  if $M \neq \emptyset$
  choose a sequence of ordered $RQP_i \in M$ based on their ranking
  decompose $q_x = RQP_i \circ \cdots \circ RQP_j \circ q'_x$
  end if
  evaluate the query by combining the results
  for all $RQP_i \cdots RQP_j \in M$
    update $\mathcal{R}(RQP_i)$
    if $\mathcal{R}(RQP_i) < Min\{\mathcal{R}(RQP)\}$
      evict the cached result of RQP_i from caching
    end if
  end for

Algorithm 5: Evolution-Conscious Cache Maintenance Policy

Input: Q, ΔQ, K be the set of queries that have been cached

Description:
  Compute $q = \frac{|\Delta Q|}{|Q|}$
  if $q \geq \varepsilon$
    Regenerate I_p and F_p.
    if $M' = K \cap I_p \neq \emptyset$
      evict $RQP \in M'$
    end if
    while there is space left in the cache
      cache the RQP with maximum rank but not in the cache
    end while
  end if

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* which we shall be using subsequently. 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: Algorithm 4 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 Algorithm 5. 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 $\varepsilon = 0.5$ 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

In this section, we evaluate the performance of the proposed mining algorithms and caching strategies with extensive experiments. 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 Microsoft Windows 2000 Professional. Two sets of experiments are conducted. The first set is to evaluate the mining algorithms for their efficiency and scalability. The second set is to compare our proposed caching strategy with the state-of-the-art frequent query pattern-based caching strategy and LRU-based cache strategy.

We use two set of synthetic datasets (queries) generated based on the DBLP.DTD[1] and SSPLAY.DTD[2]. Firstly, a DTD graph is converted into a DTD tree by introducing

[1] http://dblp.uni-trier.de/xml/dblp.dtd

[2] http://www.kelschindexing.com/shakesDTD.html

QPT ID	Time--stamp	DBLP	SSPLAY
1	day 1	dblp//key	play/act/scene
2	day 2	dblp/∗/title[author="M. Lee"]	play//stage
3	day 2	dblp/article/author	play/∗/stage
4	day 3	dblp//author	play/prologe/onstage
5	day 3	dblp/book/editor	play/act/scene/stage
6	day 5	dblp/∗/year	play//speechblock
7	day 5	dblp//page	play/epilogue/∗
8	day 5	dblp/article/author	play/prologe/onstage

Table 1: Example QPTs in the DBLP and SSPLAY Datasets

some "//" and "*" nodes. Then, all possible rooted query paths are enumerated. Similar to [8, 16, 17], 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. That is, most of the numbers of occurrences of QPTs are near the average number, while the number of very frequent and very rare QPTs is small. Example of two sets of QPTs in the DBLP and SSPLAY datasets is given in Table 1. 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 3(a).

5.1 CQP-Miner

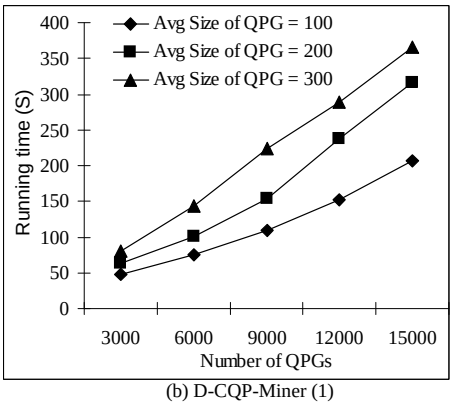
Algorithm Efficiency: We evaluate the efficiency of the algorithms by varying the average size of QPGs and the number of QPGs (the size of the time window). Figures 3(b) and (c) show the running time of the D-CQP-MINER when the size of the dataset increases. In the first case, the number of QPGs is increased 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. Note that the DBLP dataset is used and the parameters for the D-CQP-MINER 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 (Figures 3(d) and 4(a)) shows a similar trend. We use the SSPLAY dataset and the parameters for the R-CQP-MINER are fixed as follows: $\xi = 0.25$ and $\zeta = 0.05$.

Moreover, we conduct experiments to show the compactness of the two data structures we proposed. The sizes of the QPG-tree and HQPG-tree are compared with the size of the original QPTs. Figure 4(b) shows the space requirements for the DBLP and SSPLAY datasets. It can be observed that the QPG-tree and the HQPG-tree are very compact and their space requirements are approximately 50% and 70% less than those using the original data set, respectively. Relatively, HQPG-tree is much smaller than the QPG-trees as there is only one HQPG-tree for the entire query collection.

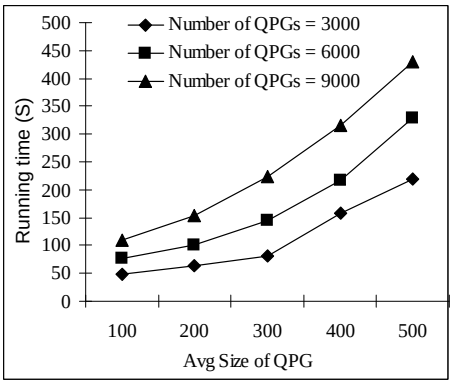
Effects of Thresholds: As there are four thresholds: α, β, γ, and ζ for the D-CQP-MINER, experiments are conducted by varying one of the thresholds and fixing

	Datasets	DBLP	SSPlay
QPT in DB	Ave # of nodes	12.4	9.5
	Max depth	10	9
	Max fanout	15	11

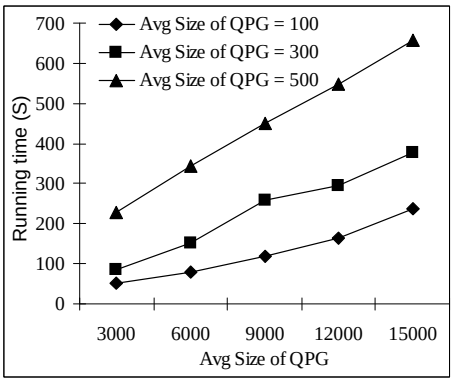
(a) Characteristics of dataset



(b) D-CQP-Miner (1)



(c) D-CQP-Miner (2)



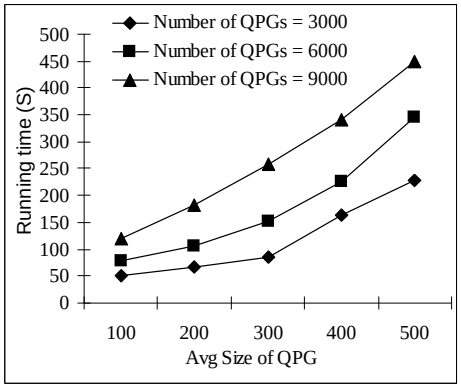
(d) R-CQP-Miner (1)

Figure 3: Datasets and perf. of CQP-MINER.

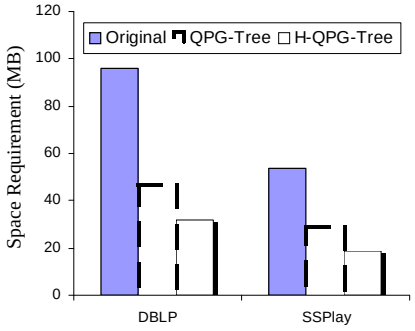
the others. For instance, in Figure 4(c), "$\alpha = 0.01 * k$, β=0.01, γ=0.02, ξ=0.1" means that we fix the values of β, γ and ξ, and vary α from 0.01 to 0.05 by varying k from 1 to 5. In this experiment, the DBLP dataset with 300,000 queries is used. The results in Figure 4(c) show that the running time of D-CQP-MINER increases when the threshold values increase. Moreover, we 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.

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(d). 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.

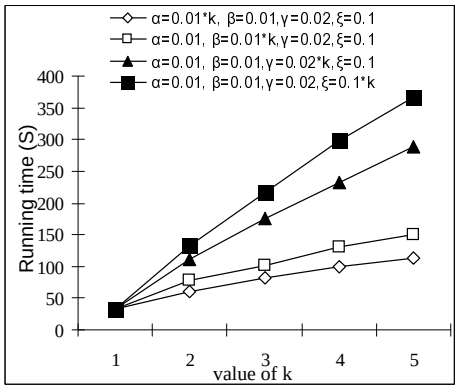
Comparison of Mining Results: As the two algorithms use different evolution metrics, to compare the mining results, we define the notion of *overlap* metric. Let F_D and I_D be the sets of FCQPS and ICQPS, respectively, in the D-CQP-MINER mining results. Let F_R and I_R be the sets of FCQPS and ICQPS in the R-CQP-MINER mining



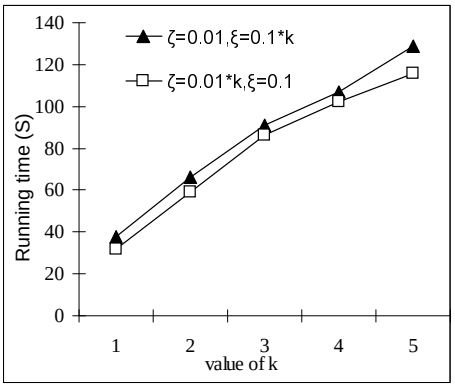
(a) R-CQP-Miner (2)



(b) Compactness of H-QPG-Tree



(c) D-CQP-Miner (3)



(d) R-CQP-Miner (3)

Figure 4: Performance of CQP-Miner (2).

results. The *overlap* between the two sets of mining results is defined as:

$$Overlap = \frac{1}{2} \times (\frac{|F_D \cap F_R|}{|F_D \cup F_R|} + \frac{|I_D \cap I_R|}{|I_D \cup I_R|})$$

.

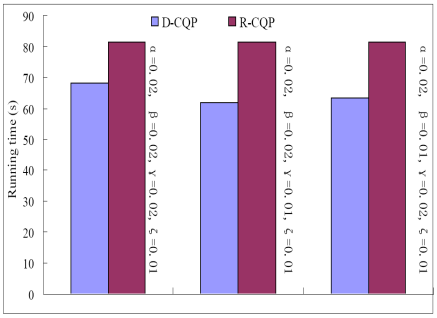
Basically, the *overlap* value is defined as the number of shared cqps divided by the total number of unique cqps in both mining results. Based on this definition, it is evident that the larger the *overlap* value, the more similar the mining results are. In this definition all the cqps in the mining 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:

$$o@k = \frac{|C_D(k) \cap C_R(k)|}{|C_D(k) \cup C_R(k)|}$$

.

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

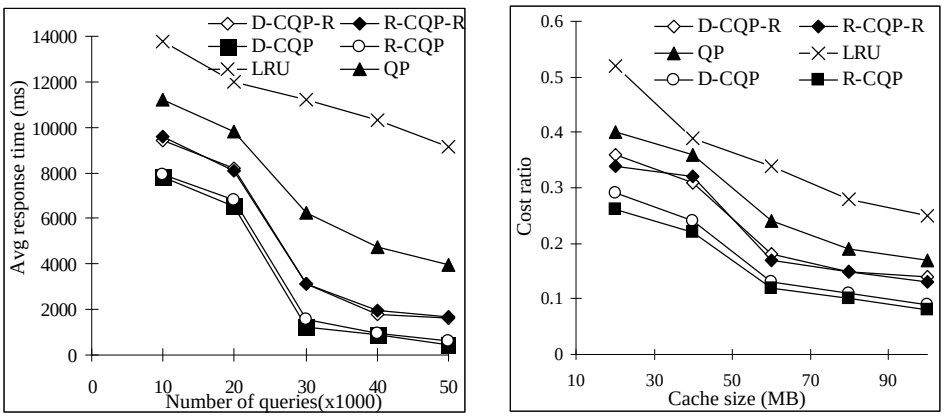
Figure 5: Comparison of mining algorithms.

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*. It can be observed that the *overlap* value depends on threshold values. 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 Table 1.



(a) Avg response time

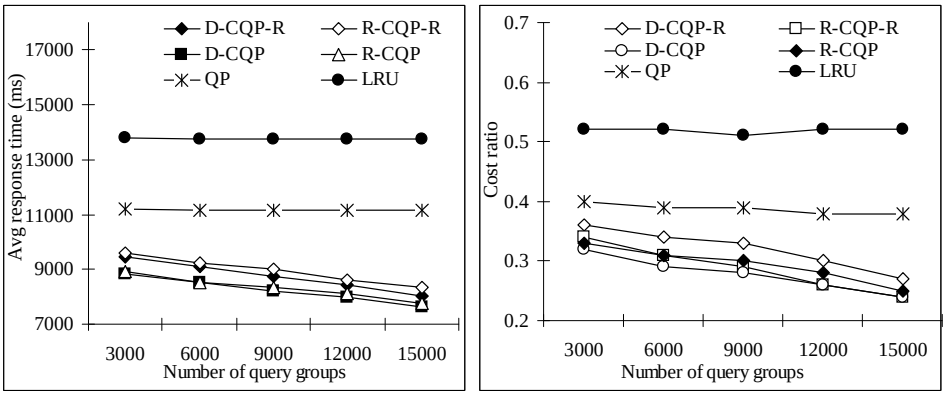
(b) Cost ratio

Figure 6: Performance of caching strategies.

We use the same storage scheme as in [16]. That is, we use the index scheme of [9] 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 [17] 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 α = 0.02, β=0.02, γ=0.01, ζ=0.01, and ξ = 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. Hence, the average response time decreases in the corresponding caching strategies. 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 out-



(a) Avg response time (b) Cost ratio

Figure 7: Effect of QPG size.

perform the rank-unconscious caching strategies highlighting the benefits of using the ranking functions. Also, the performance of the algorithms in presence of the two types of evolution metrics are almost identical. This indicates that both types of evolution metrics can improve the caching performance.

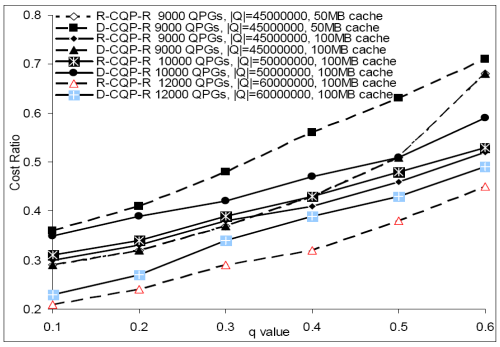
Cost ratio: The *cost ratio* is proposed to represent the query response time using different types of caching strategies against the response time without any caching strategy for all query examples. Figure 6(b) shows the performance of the six caching strategies in terms of the cost ratio measure. 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. Moreover, as the cache size increases, the caching strategies perform better but the gaps between our approaches and the existing approaches decrease.

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, thus, the caching strategies become more efficient.

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 $q < \varepsilon$ 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

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)

Figure 8: Effects of q on cost ratio.

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 $q < 0.5$ ($\varepsilon = 0.5$) . That is, our approach can improve the query evaluation performance without updating the FCQPS and ICQPS as long as $|\Delta Q| < \frac{|Q|}{2}$.

6 Related Work

XML query caching: Different caching strategies have been proposed for efficient XML query caching as traditional caching strategies may not work well for XML data [3, 7, 16, 17]. In [7], the authors proposed a compact data structure to represent the semantic regions of the XML queries and use such structures to create the remainder and refinement queries for new queries. In [3], Chen et al. proposed to use the subtype relations between two regular expression groups to tackle the XQuery containment mapping problem. Yang et al. proposed to use the frequent XML QPT to improve the XML caching strategy [16, 17]. The idea is to extract the set of QPTS that occur frequently in the query history and cache the corresponding query results for future queries. In [5, 8], incremental mining algorithms for frequent XML query patterns are proposed. Compared to these strategies, 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.

Frequent substructures mining: Most existing works in this area focus on discovering the frequent substructures from a collection of semi-structured data such as XML documents [12, 13, 18]. Our work differs from these efforts as follows. First, the frequent query pattern mining problem is different from the existing frequent subtree mining problem. Although the frequent query pattern mining and RQP mining focus only on the *rooted* subtrees/paths and the frequent subtree mining is to discover arbitrary subtrees, the first problem is more complicated than the second because of the introduction of "//" and "*" in the XML query pattern trees [16, 17]. Second, the above approaches focus only on snapshot XML data. As a result they do not focus on the discovery of novel knowledge hidden behind the historical changes to XML data. In our approach, we focus on discovering specific pattern called conserved RQP, by analyzing the temporal and evolutionary characteristics of historical XML query patterns.

Mining evolutionary patterns: Emerging pattern [6] was proposed to capture significant changes and differences between datasets. Basically, emerging patterns are defined as itemsets whose supports increase significantly from one dataset to another dataset. Our study is different from emerging pattern in that we consider the changes in a sequence of snapshots of the data while emerging pattern considers only two snapshots. In our previous work [19], we proposed a novel approach to discover the *frequently changing structures* from the sequence of historical structural changes to unordered XML. The frequently changing structures are defined as substructures that changed *frequently* and *significantly* in the history. Compared to this work, in this paper we focus on discovering XML query patterns that *do not* change frequently in the history. More importantly, compared to the above approaches, we propose a technique to build the evolution-conscious caching strategy using these patterns.

7 Conclusions and Future Work

This work is motivated by the fact that existing XML query pattern-based caching strategies ignore the temporal and evolutionary features of the historical XML queries. In this paper, we proposed a novel type of XML query pattern named conserved query paths (CQPS) for efficient caching by analyzing the evolution patterns of historical XML queries. 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. We proposed two categories of evolution metrics (delta-based and regression-based) to measure the evolutionary features of QPTS. Based on these proposed 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 D-CQP-MINER and R-CQP-MINER are efficient, scalable and can accurately identify the CQPS. More importantly, these algorithms can be effectively used to build more efficient caching strategies compared to state-of-the-art caching strategies. Furthermore, our caching approach shows good performance even when the FCQPS and ICQPS are not updated until the number of new queries issued during a specific time period is less than half of the size of historical query set.

Currently, the calendar pattern is user-defined. In future, we wish to explore how this task can be automated. Note that this is a challenging problem as different users may wish to specify different calendar patterns. 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 authors wishes to acknowledge and thank Dr Qiankun Zhao for implementing many of the ideas discussed in this report.

References

[1] S. Al-Khalifa, H.V. Jagadish, J. M. Patel et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *ICDE*, 2002.

[2] N. Bruno, N. Koudas, D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *In SIGMOD*, 2002.

[3] L. Chen, E. A. Rundensteiner, and S. Wang. Xcache: a semantic caching system for XML queries. *SIGMOD*, 618, 2002.

[4] L. Chen, S. Wang, and E. Rundensteiner. Replacement strategies for xquery caching systems. *Data Knowl. Eng.*, 49(2):145–175, 2004.

[5] Y. Chen, L. Yang, and Y-G. Wang. Incremental Mining of Frequent XML Query Patterns. *ICDM*, 2004.

[6] G. Dong and J. Li. Efficient Mining of Emerging Patterns: Discovering Trends and Differences. *KDD*, 43–52, 1999.

[7] V. Hristidis and M. Petropoulos. Semantic caching of XML databases. *WebDB*, 25–30, 2002.

[8] G. Li, J. Feng, J. Wang, Y. Zhang, and L. Zhou. Incremental Mining of Frequent Query Patterns from XML Queries for Caching. *ICDM*, 2006.

[9] Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *VLDB*, 2001.

[10] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. *VLDB*, 469–480, 2005.

[11] R. Ramesh and L.V. Ramakrishnan. Nonlinear Pattern Matching in Trees. *JACM*, 39(2):295-316, 1992.

[12] A. Termier, M-C. Rousset, and M. Sebag. TreeFinder: a First Step towards XML Data Mining. *ICDM*, 2002.

[13] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. *PAKDD*, 2004.

[14] S. Weisberg. *Applied Linear Regression*. Wiley, 2 edition, 1985.

[15] L. Yang, M. Lee, W. Hsu, and S. Acharya. Mining frequent query patterns from XML queries. *DASFAA*, 355–362, 2003.

[16] L. Yang, M. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. *VLDB*, 69–80, 2003.

[17] L. Yang, M. Lee, W. Hsu, and X. Guo. 2pxminer: an efficient two pass mining of frequent XML query patterns. *SIGKDD*, 731–736, 2004.

[18] M. J. Zaki. Efficiently mining frequent trees in a forest. *SIGKDD*, 71–80, 2002.

[19] Q. Zhao, S. S. Bhowmick, M. Mohania, and Y. Kambayashi. Discovering Frequently Changing Structures from Historical Structural Deltas of Unordered XML. *CIKM*, 2004.