Load Awareness in FliX

by

Mohammad Abdulmalik Al-Rifai

Thesis

presented to the Faculty of Computer Science

At University of the Saarland

in partial fulfillment of the requirements for the degree of

Master of Science

September 2005
Ich erkläre hiermit an Eides statt, da ich die vorliegende Arbeit vollständig selbst und nur unter Verwendung der im Literaturverzeichnis angegeben Quellen erstellt habe.

Saarbrücken, im September 2005
Abstract

The rapid growth of XML repositories over the past several years and using XML as a universal data exchange format between heterogeneous systems and in the World Wide Web, poses interesting challenges to database researchers, namely, managing and querying large collections of XML documents.

While there are many proposals for path indexes on XML documents, none of them is perfectly suited for indexing large-scale collections of interlinked XML documents.

FliX (Framework for indexing large collections of interlinked XML documents) [1] was developed to support an efficient incremental evaluation of descendants’ queries on large, heterogeneous, strongly linked document collections, using existing path indexing strategies as building blocks. By studying and analyzing the performance of the current implementation of this framework, this thesis aims to improve the query evaluation efficiency in FliX. To achieve this, a theoretic model for the cost of evaluating a collection of queries is built and used along with statistical information to minimize the total cost of a given query load. FliX is extended by integrating a special-purpose load-aware caching unit into its current implementation. A complete implementation and evaluation of the extended version of FliX is provided.
# Table of Contents

## Acknowledgments

## Chapter 1: Introduction

1.1 XML Data Model ................................................................. 1  
1.2 Problem Statement ............................................................. 5  
1.3 Existing Solutions ............................................................... 7  
1.4 FliX ............................................................................... 12  
1.5 Contribution and Outline ....................................................... 14

## Chapter 2: FliX

2.1 The Concept ....................................................................... 17  
2.2 FliX Architecture ................................................................. 20  
2.3 Path Expression Evaluation in FliX ....................................... 21

## Chapter 3: Load-aware Caching in FliX

3.1 Formal Model and Notation .................................................. 27  
  3.1.1 Query Load ................................................................. 27  
  3.1.2 Evaluation Cost ............................................................ 28  
  3.1.3 Replacement Problem ................................................... 31  
3.2 Caching Models .................................................................. 32  
3.3 Existing Replacement Algorithms ......................................... 32  
3.4 Caching in FliX ................................................................. 35  
  3.4.1 Caching Benefit Estimation ............................................ 35  
  3.4.2 Caching Optimization ................................................... 36  
  3.4.3 Caching Algorithm ....................................................... 39

## Chapter 4: Implementation

4.1 Extended FliX Architecture ................................................... 43  
4.2 Cache Structure ................................................................. 44
Acknowledgments

At the end of my thesis I would like to thank all those people who made this thesis possible. I would like to thank Prof. Gerhard Weikum for his support and for giving me the opportunity to join his very active group.

I am very grateful to my advisor Dr. Ralf Schenkel who guided this work and helped whenever I was in need.

Finally, I would like to express my deepest gratitude for the constant support, understanding and love that I received from my wife and my mother during the past years.
Chapter 1
Introduction

1.1 XML Data Model

XML[3] (Extensible Markup Language) is an abbreviated version of Standard Generalized Markup Language (SGML), for the exchange of structured documents over the Internet. Unlike HTML, XML readily enables the definition, transmission, validation, and interpretation of data between differing computing platforms and applications. XML permits people in a specialized field, such as chemistry, finance, or environmental data collection to develop XML schema that define the markup language for the exchange of specialized data unique to their fields.

XML is extensible, meaning a developer can extend the language by devising new tags to describe and share data in any specialized way desired as long as the new tags follow the XML syntax defined by the World Wide Web Consortium (W3C) XML specifications. XML is very useful for organizations that do not share but need to develop a common data exchange format. Its extensibility provides flexibility in developing exchange formats in XML schema, provided all partners agree on the data format and definitions of the data it contains.

To express the relationships between elements and the nested structures in an XML document, XML documents are often modelled as trees. Tree nodes represent document elements, attributes or text data, while edges represent the element/sub-element (or parent/child) relationship. In addition to the hierarchical structure of elements, XML supports intra–document links by using IDREFs [13], and inter–document links by using XLinks [14]. XLinks are used for creating both basic unidirectional links (like HTML’s hyperlinks using anchor tags) and more complex linking structure. The usage of links in XML documents turns the model into an unweighted (possibly cyclic) directed graph model. Figure 1 shows two interlinked XML documents and their corresponding graph representation.
ResearchGroup.xml:

```xml
<Members>
  <Member member_id="H1" type="head">
    <FirstName>Gerhard</FirstName>
    <LastName>Weikum</LastName>
  </Member>
  <Member member_id="R1" type="researcher">
    <FirstName>Ralf</FirstName>
    <LastName>Schenkel</LastName>
  </Member>
  ...  
  <Student member_id="S1">
    <FirstName>Mohammad</FirstName>
    <LastName>Alrifai</LastName>
    <ThesisTitle>Load Awareness in FliX</ThesisTitle>
    <Supervisor member_id="H1"/>
    <Supervisor member_id="R1"/>
  </Student>
</Members>
```

Courses.xml:

```xml
<Courses>
  <Course course_id="C1">
    <Title>XML Technologies</Title>
    <Lecturer xlink:type="simple"
      xlink:href="ResearchGroup.xml#xpointer(id('R1'))"
      xmlns:xlink = "http://www.w3.org/1999/xlink/namespace"
    />
  </Course>
  ...  
</Courses>
```

Figure 1.1-(a): Two XML Documents (ResearchGroup.xml and Courses.xml) with some inter- and intra-link
Based upon the above discussion, we present the following definitions:

For each XML document $d$, we consider the element-level tree $T_E(d) = (V_E(d), E_E(d))$ where the vertex set $V_E(d)$ consists of all elements of $d$ and the edge set $E_E(d)$ represents all parent-child relationships between elements in $d$. Additionally, we maintain the set $L_I(d) \subseteq V_E(d) \times V_E(d)$ of all intra-document links within $d$. The element-level graph $G_E(d) = (V_E(d), E_E(d))$ has the same node set as the tree, but its edge set is extended by the intra-links, i.e. $E_E(d) = E'_E(d) \cup L_I(d)$.

A collection $X = (D, L)$ of XML documents consists of a set $D = \{d_1, \ldots, d_n\}$ of documents together with (a subset of) the set $L$ of inter-document links between documents in $D$, i.e., pairs of elements from different documents that are connected.

**Figure 1.1-(b):** a directed graph representing the two XML documents in 1.1-(a)
via a link. The element-level graph $G_E(X) = (V_E(X), E_E(X))$ for a collection $X = (D, L)$ of XML documents has as vertex set the union of the elements of the documents in $D$, and as edge set the union of the edge sets of the element-level graphs for the documents plus the set $L$ of inter-document links. The document mapping function $\text{doc}: V_E(X) \rightarrow D$ for a collection $X = (D, L)$ maps vertices of the element-level graph of the collection to the document they originate in.

To query XML documents, several XML query languages have been proposed in the literature. Examples are Lorel [3], XML-QL [4], XML-GL [5], Quilt [6], XPath [7], and XQuery [8]. XQuery that has been developed by the World Wide Web Consortium (W3C) is being standardized as a major XML query language. The main building block of XQuery is XPath, which addresses part of XML documents for retrieval, both by value search and structure search. Using XPath’s regular path expressions it is possible to navigate through arbitrary long paths in the hierarchical structure of an XML document, which is like using path steps in URLs. Starting from a context node, an XPath query traverses its input document using a number of steps. A step’s axis indicates which tree nodes are reachable from the context node, the step’s node test then filters the reachable nodes by tag name or node kind. These intermediary nodes are then, recursively, interpreted as context nodes for subsequent steps, and so forth. The XPath specification [12] lists a family of 13 axes (see Table 1.1), among these the child and descendant-or-self axes, probably more widely known by their mnemonic abbreviations / and //, respectively.

For example, //book//Chapter is a path expression that corresponds to finding all Chapter elements that are contained in a book element. To evaluate the query //book//Chapter, a naive tree traversal strategy could cause a scan of the whole XML data tree even when there are only few results. Alternatively, the set-at-a-time strategy would first retrieve all book and Chapter elements, possibly with some index on the labels of the elements, and then find all occurrences of the ancestor-descendant relationship between the element sets.
1.2 Problem Statement

It is expected that XML will become the lingua franca of the Web, eventually replacing HTML. This implies the capability of hyperlinking XML sources in the same way, as HTML documents are currently interlinked. Therefore, the W3C has recently provided a powerful means for interlinking XML documents, namely, XML Linking Language (XLink) and XML Pointer language (XPointer). XLink provides a framework for creating both basic unidirectional links and more complex linking structures. Compared with relational databases and its classical query languages (e.g. SQL), the new concept of having links inside the data is new and poses new challenges for querying such documents. This is simply because relational databases and SQL do not support recursion, which a naïve approach would suggest for query evaluation.
It is widely believed that the current state of the art of the relational database technology fails to deliver all necessary functionalities of storing and querying XML documents efficiently. Therefore an XML query engine that is based on a relational database for storing XML documents needs to have an efficient path index that stores paths between elements of XML documents. This index is typically used to quickly evaluate path expressions like \(//A//B\) where the conventional alternative would be to traverse the hierarchy of the XML data, which can yield a substantial overhead.

The ability to effectively rank retrieved results in order of their expected relevance to a query is an important factor in retrieval systems. This is especially important as users are typically not interested in seeing all results of a query (which often would be way too many with documents from the Web), but may be satisfied with the top \(k\) results, with \(k\) usually less than 100. In the context of path expressions, results ranking means returning results in ascending order with respect to the distance between sources and targets.

In this Master's thesis we are focusing on the problem of finding an efficient path index that can achieve the following goals effectively in the context of large collections of inter-linked XML documents:

1- evaluate path expressions like \(//A//B\), i.e., find all pairs \((a,b)\) in ascending order of distance between \(a\) and \(b\), where \(a\) is an element of type \(A\) and \(b\) is an element of type \(B\) and \(b\) is either a local or a remote descendant of \(a\). By ‘local’ descendants we mean descendants within the same XML document, that the ancestor belongs to, and by ‘remote’ descendants we mean descendants within different XML documents than the one the ancestor belongs to.

2- evaluate queries like \(a//B\), i.e., find all elements \(b\) of type \(B\) that are either local or remote descendants of the specific element \(a\), again ordered in ascending order of distance between \(a\) and \(b\).
1.3 Existing Solutions

Several different approaches for indexing path information have been proposed, from very naive ways to highly efficient and space-effective index structures. While they are typically quite efficient in evaluating path expressions, these approaches widely differ in space utilization, support for paths with wildcards, support for links and support for updates. In this Section we explore some of the well-known existing solutions.

1.3.1 Pre/Postorder Scheme (PPO):

This path index proposed by Grust [21] computes the preorder and the postorder for each element of a single XML document without links, by traversing the document in depth-first order. In a preorder traversal, a tree node $e$ is visited and assigned its preorder rank ($\text{pre}(e)$) before its children are recursively traversed from left to right. In a postorder traversal, a node $e$ is assigned its postorder rank ($\text{post}(e)$) after all its children have been traversed from left to right. In Figure 1.2-(a) the preorder (on the left-hand side) and the postorder (on the right-hand side) of each node on the tree is shown. Both values are stored for each node in the document tree during the build phase. Thus, building the index takes time $O(|EX|)$, and space consumption is $O(|VX|)$. In Figure 1.2-(b) the distribution of all nodes of the document on a pre-order/post-order plane is shown. To support all XPath axes it is sufficient to keep track of the parent’s preorder rank $\text{parent}(e)$, see Table 1.2. For example, there is a path from $x$ to $y$ iff $\text{pre}(x) < \text{pre}(y)$ and $\text{post}(x) > \text{post}(y)$. This index structure is highly efficient for reachability queries and can, with slight additions, also provide the distance between two nodes. However, this path index does not support links between elements and requires recomputing the preorder and postorder of nodes after each insertion/removal operation.
Figure 1.2: (a) XML tree with each node assigned a preorder rank (to the left) and a postorder rank (to the right); (b) the main XML document paths as seen from the context node $f$.

<table>
<thead>
<tr>
<th>Axis</th>
<th>preorder</th>
<th>postorder</th>
<th>parent</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>child</td>
<td>$(\text{pre}(v), \infty)$</td>
<td>$[0, \text{post}(v))$</td>
<td>$\text{pre}(v)$</td>
<td>*</td>
</tr>
<tr>
<td>descendant</td>
<td>$(\text{pre}(v), \infty)$</td>
<td>$[0, \text{post}(v))$</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>$[\text{pre}(v), \infty)$</td>
<td>$[0, \text{post}(v))$</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>parent</td>
<td>$[\text{par}(v), \text{par}(v)]$</td>
<td>$(\text{post}(v), \infty)$</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ancestor</td>
<td>$[0, \text{pre}(v))$</td>
<td>$(\text{post}(v), \infty)$</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>$[0, \text{pre}(v))$</td>
<td>$[\text{post}(v), \infty)$</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>following</td>
<td>$(\text{pre}(v), \infty)$</td>
<td>$(\text{post}(v), \infty)$</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>following-sibling</td>
<td>$(\text{pre}(v), \infty)$</td>
<td>$(\text{post}(v), \infty)$</td>
<td>parent$(v)$</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 1.2: XPath axes and the corresponding result windows given a context node $v$. 
1.3.2 DataGuides:
DataGuides have been proposed by Goldman & Widom at VLDB97 [8]. They can be perceived as dynamic schemas, but are often extended with targeted node identifiers to serve as index. Such indexes are called targeted DataGuide. DataGuide can also help in query formulation. They are rather concise and accurate structural summaries. Every path in the database has one and only one corresponding path in the DataGuide with the same sequence of labels. A rooted label path of a node is a sequence of labels defining a path from the root to the node. To achieve conciseness, a DataGuide describes every unique rooted label path of a collection exactly once. To ensure accuracy a DataGuide encodes no rooted label path that does not appear in the collection. DataGuides for a given collection of documents are in general not unique.

A strong DataGuide is such that any indistinguishable rooted label paths \( p \) and \( p' \) on the DataGuide \( G \) are also indistinguishable on the database \( DB \); i.e. \( p(G) = p'(G) \Rightarrow p(DB) = p'(DB) \). A strong DataGuide is unique for a given collection of XML documents. If we view \( DB \) as an automaton by making each node a terminal state and each root an initial state, then a DataGuide is by definition a deterministic automaton equivalent to \( DB \). Among all DataGuides only one has states, which are related to sets of nodes in \( DB \); this is called a strong DataGuide, and is precisely the standard powerset automaton of \( DB \). The building algorithm of the strong DataGuide emulates the conversion algorithm from the nondeterministic finite automaton (NFA) to the deterministic finite automaton (DFA) [16]. This conversion takes linear time for tree-structured data and exponential time in the worst case for graph-structured data. Furthermore, on very irregularly structured data, the strong DataGuide may be much larger than the original data.

Targeted DataGuides store for each path node identifiers corresponding to elements having the given path labels. The index is thus an unordered graph with a list of document and node identifier for each path of length 1 to \( N \). DataGuides are difficult to maintain for documents with cycles as paths should be limited in size. For
searching on keywords, nothing is provided and an additional value index shall be maintained as in Lore [3].

Moreover, DataGuides store only paths starting from the root in the data graph, therefore, they are inefficient for queries with partial matching expressions as the query processor has to rewrite partial matching path queries into queries with simple path expressions by the exhaustive navigation of index structures, thus resulting in performance degradation.

1.3.3 The Index Fabric (Indexing Rooted Paths with Values)
The index Fabric has been introduced by [18]. It includes both an efficient implementation of the DataGuide and a clever extension with values in place of identifiers. To shorten the paths, labels are first encoded with one (or more) letter. All paths from the root to a leaf containing data are prearranged as a sequence of encoded labels followed by the value as a string. For example, the path /University/Address/City[‘Berlin’] is encoded as UACBerlin. To store the encoded strings, the method uses an efficient index for strings, i.e., a Patricia trie. A Patricia trie is a simple form of compressed trie that merges single child nodes with their parents. A balancing mechanism is added to the Patricia trie to guarantee constant access time when searching for paths of length $N$. Path expressions including predicates on values for elements are performed as string search. Notice that the index does not keep information on non-terminal nodes. It does not manage node identifiers either. Also, it does not keep the order of elements in documents although it stores terminal node values. Moreover, the Index fabric cannot process queries with wildcards in path expressions efficiently.

1.3.4 APEX (Indexing Frequently Used Paths)
While the strong DataGuide maintains all paths from the root, APEX [19] does not keep all paths but utilizes frequently used paths to improve query performance. APEX maintains in the index all paths of length 1 plus additional required paths, i.e., the set of labels plus some composed paths, those that are frequently queried. An APEX
index consists of two structures, a graph structure representing the structural summary of the data, and a hash tree structure that associates required paths to nodes of the graph structure. For efficiency, each node of the hash tree is implemented as a hash table on the labels. Terminal nodes refer to a graph structure node. The hash tree is used to find nodes of the structure graph for given label path, also for incremental update. A graph structure node contains the list of node identifiers of the XML documents, similarly to a targeted DataGuide node. The APEX graph structure is simpler than the DataGuide and accessed through the hash tree. Nice algorithms are proposed to perform incremental updates of the hash tree and graph structure. APEX is efficient if all paths are short and don’t contain wildcards (arbitrary long path query, like X//Y).

1.3.5 HOPI:
HOPI is a connection index [22] that makes use of a compact representation of reachability and distance information in graphs proposed by Cohen et al. [23]. The basic idea of HOPI is to compute so-called 2-hop labels for each node of the whole XML graph. Let $T = \{(u, v) \mid \text{there is a path from } u \text{ to } v \text{ in } G\}$ the set of all connections in a directed graph $G = (V,E)$ (i.e., $T$ is the transitive closure of the binary relation given by $E$). For each connection $(u, v)$ of $G$ (i.e., $(u, v) \in T$) choose a node $w$ on a path from $u$ to $v$ as a center node and add $w$ to a set $\text{Lout}(u)$ of descendants of $u$ and to a set $\text{Lin}(v)$ of ancestors of $v$. Now testing if there is a path between two nodes $u$ and $v$ can be done efficiently by checking if $\text{Lout}(u) \cap \text{Lin}(v) \neq \emptyset$. There is a path from $u$ to $v$ iff $\text{Lout}(u) \cap \text{Lin}(v) \neq \emptyset$; and this connection from $u$ to $v$ is given by a first hop from $u$ to some $w \in \text{Lout}(u) \cap \text{Lin}(v)$ and a second hop from $w$ to $v$, hence the name of the method, see Figure 1.3.

The index maintains for each element $e$ two sets of elements (so-called labels) $\text{Lin}(e)$ and $\text{Lout}(e)$ such that there is a path from $x$ to $y$ iff $\text{Lout}(x) \cap \text{Lin}(y) \neq \emptyset$. These labels can also be augmented with distance information to compute the distance of two elements. HOPI provides a divide-and-conquer algorithm for index creation to partition the whole XML graph into subgraphs such that subgraphs have a small
number of links across each other. This algorithm is reasonably fast as long as the document collection is not too large. Querying the index for reachability of two elements is very fast for most elements. Experiments [22] indicate that the HOPI index is usually an order of magnitude more compact than the transitive closure for document sets with relatively few links, and even more compact if the number of links increases. However HOPI’s size may grow large for large documents sets, when merging covers for the partitions (which may increase the size of HOPI index). An important issue is the time to build HOPI index, which highly increases with increasing number of documents.

![Two-Hop Cover](image_url)

**Figure 1.3:** Two-Hop Cover

### 1.4 FliX

None of the aforementioned path indexes is perfectly suited for indexing large-scale collections of interlinked XML documents. All of the proposed path indexes lack the support for intra-document links (using ID and IDRF attributes) and inter-document links (using XLink and XPointer links).

The problems that arise when taking links into account are twofold:

- **Intra-document links** change the structure of XML documents; i.e. they are no longer trees, but form an (arbitrary) graph. This makes some of the highly efficient path indexes like pre/postorder fail. Such path indexes that are tailored to index tree-like XML documents cannot even be extended to handle such graphs.

- **Inter-document links** generate interconnections of previously unconnected XML documents, yielding a small number of very huge connected sets of XML documents that should be indexed as one (meta) document. However, given that
all indexing strategies have nonlinear space complexity, this can hardly be applied to Web-scale document collections, where millions of interlinked documents with thousands of elements in each document would result in several Tera-, if not Petabytes of index storage, and the time needed to build the index would exceed all limits.

To cope with these two problems, FliX (Framework for indexing large collections of interlinked XML documents) [1] was proposed. FliX takes the whole collection of XML documents as input and divides it into many partitions (meta-documents) during its building phase. After partitioning the collection, FliX selects the ‘best’ path index for each partition depending on its nature and structure. As an example, for the given set of partitions shown in Figure 1.4, FliX will use per/postorder index for Meta-document 1, since it forms a tree whereas on the hand for the strongly interlinked partition Meta-document 2, pre/postorder path index cannot be used, therefore HOPI would be a good option. FliX maintains links between both partitions in Figure 1.4 separately.

![Figure 1.4: Two interlinked Meta-Documents](image)

In the query evaluation phase, results are computed incrementally. When FliX gets a query as input, it firstly maps the source of the query to its partition and then by means of the local path index of that partition it evaluates the query and returns the results to the client. In the next step, FliX uses the set of links between the partitions to find if there is any outgoing link from the partition of the originating source into other partitions. If there is any such link, FliX uses them to find all other partitions.
that are reachable from the originating source. The local indexes in the reachable partitions are then used to find possible descendants that qualify for the given query. Again, FliX looks for any further partitions reachable from the current ones and the same procedure is repeated until either there are no more reachable partitions or all targeting partitions have already been visited.

The experimental results in [2] show that FliX outperforms all of the proposed solutions in terms of evaluation time and index size, and is therefore more scalable for indexing large collections of XML documents. However, the performance of FliX could be further improved by employing a smart caching system that helps to answer frequent queries very quickly, thus reducing the total cost that is needed to evaluate a large collection of queries.

1.5 Contribution and Outline

The Path Expression Evaluator in FliX is not optimized to handle a large collection of queries that are built upon a large amount of path expressions, where most of them are frequent or in many cases some of the path expressions share the same originating source (i.e. ancestor). This is likely the case in some applications where most of the ancestor-descendant path expressions often start with a small set of possible elements. For instance, queries like $\text{author}/*$ that share the same ancestor ‘author’ (e.g. $\text{author}//\text{book}$, $\text{author}//\text{article}$, ...etc) or queries like $\text{proceeding}/*$ that share the ancestor ‘proceeding’ (e.g. $\text{proceeding} //\text{year}$, $\text{proceeding} //\text{title}$, ...etc) are very frequently used to retrieve data out of a digital library like DBLP.

In this Master’s thesis, we aim to improve the performance of the query evaluation in FliX and optimize it for efficiently evaluating a large collection of queries by reducing the total time needed to evaluate the whole collection of queries. To achieve this goal we extend FliX by encapsulating a special-purpose cache into the framework. The newly extended block should cache both results of recently evaluated queries and their complete tree of intermediate sources (i.e. entry nodes to descendant
partitions). This enables FliX to answer not only previously seen queries but also queries that share the same tree of intermediate sources, i.e. queries that share the same prefix (ancestor). As such, caching in FliX is different from the traditional tuple or page-based caching systems, since it exploits the idea of reusing cached information to answer new queries.

Additionally, caching in FliX maintains meta-information like the distance between the source of the query and each of its result nodes and intermediate-sources. This enables FliX to keep the distance-based ranking of the results when a request is answered from the cache. By keeping the distance information of the intermediate-sources, while answering new requests for further distance results, FliX is still able to keep the rank of the new results by using the distance of the intermediate-sources as an offset.

Caching in FliX adopts a replacement policy that is based upon an estimation function that exploits statistical information about the query load. This estimation function attempts to estimate the benefit of keeping an entry in the cache and the benefit of caching a new entry, and based on these computations a decision about the replacement is made. This approach enables the load awareness in FliX.

The rest of this Master’s thesis is organized as follows:

- In Chapter 2 we explore FliX framework in details by introducing its architecture and explaining the query evaluation procedure. We also discuss the performance of FliX in the context of evaluating large collections of queries.

- In Chapter 3 we introduce a theoretical model that models the cost of evaluating a collection of queries formally. The proposed caching algorithm is also provided.

- In Chapter 4 we describe the implementation of the load-aware caching in FliX.
In Chapter 5, all experiments that were carried out together with their results are presented and evaluated to show how the performance of query evaluation in FliX is significantly improved.

Finally, in Chapter 6 we summarize our work and the outcome of it.
Chapter 2

FliX: Framework for indexing large collections of interlinked XML documents

In this Chapter we introduce FliX as a flexible framework for efficiently indexing and querying large collections of inter-linked XML documents like web portals and digital libraries, for example. First, we explain the basic ideas behind this framework, then we introduce its architecture and finally we analyze and evaluate its performance with respect to its query evaluation algorithm. At the end of this Chapter we show how query evaluation in FliX can be significantly improved and optimized by modifying the current algorithm and implementing a clever caching strategy.

2.1 The Concept

FliX adopts the “divide and conquer” strategy as a simple solution. Firstly the whole graph that represents the collection of XML documents is partitioned into many small and manageable partitions, called meta-documents. A meta-document consists of a set of inter-linked XML documents. This is done by the so-called Meta Documents Builder in the Build Phase. Links (both inter- and intra-document links) that are not contained in any meta-document (i.e. links between meta-documents) are separately maintained. After that FliX attempts to reuse existing indexing strategies as building blocks by selecting the “best” path index for each meta-document given its characteristics. Thus, FliX is extensible and can easily support all known (or even yet to be proposed) path indexing strategies for indexing the subsets of the collection of XML documents. It also can be tailored to the needs of the application and to the
structure of the data to be indexed. FliX efficiently supports XPath’s descendants-or-self axis, and can handle other query types, too.

In the *Query Evaluation Phase*, a given path expression is evaluated incrementally by the *Path Expression Evaluator* of FliX. Starting from some given meta-documents, the *Path Expression Evaluator* uses the local indexes in the corresponding meta-documents to answer the path expression. After that, and by means of the maintained list of links between meta-documents, the *Path Expression Evaluator* of FliX proceeds by traversing the outgoing links towards the next meta-documents in the graph (which are pointed to by current meta-documents) that might contain some potential candidates.

![Figure 2.1: Path Expression Evaluation process in FliX](image-url)
An example of the path expression evaluation process is depicted in Figure 2.1, where the different steps of the process are labeled and numbered. To evaluate a query that contains a path expression like \(x//Y\) (which means to find all elements with tag name ‘Y’ in the whole graph that are, both directly and indirectly, descendants of the starting node ‘x’), a client (e.g. a search engine) sends the path expression to the framework over the user interface. The framework then sends the specified path expression to its Path Expression Evaluator (PEE) and creates a channel (pipeline) between the PEE and the client. The PEE incrementally evaluates the given path expression and sends the resulting nodes as soon as it gets them on the pipeline to the client. In the evaluation process local index of the same meta-document of the starting element (node ‘x’ in our example) is firstly used to answer the query. Descendant elements of the type ‘Y’ that are in the same meta-document are returned quickly to the user. After that, starting from the originating meta-document, FliX traverses the outgoing links looking for farther descendants that probably reside in other meta-documents. This procedure is repeated until either there is no more outgoing links to traverse or the maximum required distance of the results is reached. Whenever a new meta-document is encountered during this process, its local index is used to return any potential descendants of the type ‘Y’. Results are always returned directly to the client over the pipeline for further processing in an approximate ascending order in terms of distance between the ancestors and their descendants. A search engine using FliX for indexing can therefore return the best results early to the user and may even stop the execution when it can determine that it has produced the top \(k\) results (e.g., using an algorithm similar to Fagin’s threshold algorithm [9] with only sequential reads), or alternatively when the user decides to stop the query.
2.2 FliX Architecture:

The architecture of FliX is shown in Figure 2.2 and consists of the following core components:

- Several *Path Indexing Strategies (PIS) S₁, . . . , Sₙ* among them PPO, APEX and HOPI, that support the XPath axes and return results in ascending order of distance.

- The *Meta Document Builder MDB* that automatically builds an “optimal” set of *meta documents M = {M₁, . . . , Mₘ}* where each meta document consists of a distinct subset of the set X of interlinked XML documents to be indexed.

- The *Indexing Strategy Selector ISS* that automatically selects, for each Mᵢ of the meta documents, the “optimal” indexing strategy Sᵢ, based on structure, size and other properties of the meta documents.

- The *Index Builder IB* that builds index structures I₁, . . . , Iₘ for each of the meta documents, using the automatically selected indexing strategy for each meta document.

- The *Path Expression Evaluator PEE* that evaluates path expressions with the descendants-or-self axis and returns results either approximately or exactly ordered by ascending path length.

The Meta Document Builder, the Indexing Strategy Selector, and the Index Builder are used in the *Build Phase*, while the Path Expression Evaluator is used in the *Query Evaluation Phase*. 
In [2], a first and complete implementation of FliX was presented and a number of different experiments results were shown. The performance of FliX in general was evaluated. It was also shown how FliX is able to efficiently answer queries that include path expressions like ancestor-descendant expressions.

The concept of the Query Evaluation in FliX was presented in Section 2.1. In this Section we aim to give a closer look into the incremental path expression evaluation procedure of the PEE in FliX in order to be able to Figure out how this very important building block of FliX can be enhanced.

In Figure 2.3 the algorithm of the path expression evaluation in FliX is shown. According to this algorithm, to evaluate a path expression like x//Y, a priority queue is used which is initialised by inserting the starting node ‘x’ and setting its priority (distance) to zero. In line 8 the corresponding path index (IND) of the meta-document
(M) that holds ‘x’ is determined. This local path index is then used in line 9 to get all descendants of node ‘x’ in the specified meta-documents that are of type element with tag name ‘Y’. All possible resulting nodes are returned immediately to the client over the pipeline. After evaluating all qualified descendants in M, the local index IND is used again (together with another index of all inter-meta-documents links) to get a set (LE) of reachable entry-elements that belong to other meta-documents. Each resulting entry node is then inserted into the priority queue with a priority equal to its distance from the starting source node ‘x’. Previous steps are repeated for each entry in the priority queue as the newly inserted elements are considered as the sources of the next iteration. The process continues till there are no more sources (entry nodes) inserted into the queue.

```plaintext
1  Stream PEE(Element x, Type Y)
2  PriorityQueue Q;
3  Q.insert(x,0);
4  while (Q not empty)
5      do
6          Element e=Q.extractMin();
7          MetaDocument M=e.meta();
8          Index IND=M.Index();
9          Set R=IND.findDescendantsByName(e,Y);
10         for all r in R
11             do
12                return streamed r;
13          od
14         od
15         Set LE=IND.getReachableEntryNodes(e);
16         for all l IN LE
17             do
18                 Q.insert(l,e.dist()+IND.dist(e,l));
19             od
20         od
```

**Figure 2.3:** Path Expression Evaluation Algorithm of PEE in FliX

The bottleneck in the given algorithm are those steps in line 9, and 14, namely, finding all qualified descendants in the specified meta-documents and finding all
reachable *entry nodes* (white-colored nodes in Figure 2.1) in other meta-documents respectively. While the former is completely dependent on the local path index in the particular meta-document, the latter is not. Finding all entry nodes in meta-documents that are reachable from a given element in a given meta-document implies two different steps. At first, all descendants in the same meta-documents that have some outer-links, i.e. links pointing to nodes outside their own meta-document, are computed. This first task is done by means of the local path index. Secondly, a special index which is maintained by FliX for indexing links between meta-documents that are not indexed by any local path index is used to retrieve targeting entry nodes in other meta-documents. The latter task is time consuming, which significantly impacts the total path expression evaluation procedure in FliX.

As part of this thesis, we ran some experiments to find out how much does finding the next reachable entry nodes contribute in the cost of the path expression evaluation in FliX. We used the framework to evaluate a collection of 40 different queries. Statistical information like the time consumed in different steps of the evaluation process was collected.

![](image)

**Figure 2.4:** the cost of finding next entry elements in PEE
As shown in Figure 2.4, the results of experiments clearly prove that finding next entry elements during the path expression evaluation is very costly with respect to the total evaluation time. In average, this takes around 40% of the total time of evaluation. For some queries, the percentage of this cost may even reach 70-80% of the total time. These results are actually reasonable, since, as we described earlier in this Section, finding the next entry elements implies both finding a set of all descendant elements in the same meta-document and then joining this set with the set of links between meta-documents.

Because duplicated entry nodes are immediately eliminated during the evaluation process, it is obvious that the complete set of entry nodes that are reachable from the starting source node form a tree. This tree is rooted at the starting source node and its depth depends on the maximum required distance of results that is set by the client at the beginning of the query evaluation process. Figure 2.5-b shows all the meta-documents that are traversed during the different rounds of evaluating a descendant-or-self path expression like x//Y. In Figure 2.5 the resulting tree of all targeting entry elements that are reachable from source x is depicted.
Figure 2.5-a: meta-documents reachable from \( x \) are traversed during evaluation steps

Figure 2.5-b: dotted arrows represent edges in the tree of entry elements rooted at \( x \)
An interesting observation that we made during the experiments is the following: for some $n$ queries (ancestor-descendant path expressions) that originate from the same source elements (for example: $x/Y$ and $x/Z$), the same set of entry elements is incrementally computed $n$ times, with the same total cost being paid $n$ times; i.e. the whole graph is traversed $n$ times to find the same tree of entry elements rooted at $x$. However, this could be avoided if the nodes of this tree together with all necessary information (like their distance from $x$ and the id of meta-documents that hold each of them) were kept for future use. In such case, a new query that starts from node $x$ can be evaluated in one bulk step, since the local indexes that hold the cached entry elements can be used concurrently at one step to find possible descendants in their meta-documents.

Moreover, a client may set a query along with a maximum distance required for resulting nodes. In some cases a client may stop the evaluation process at some point before the Path Expression Evaluator has traversed the whole graph. Now in the same session, the client may refine the previously asked query by increasing or decreasing the required maximum distance specified in the last action. The current implementation of FliX consider the refined query as a new one and answers this query by re-traversing the whole graph, which could be avoided if the intermediate sources (i.e. entry nodes) computed in the last step of the last evaluation of the query were cached.

The new extended version of FliX that we introduce in Chapter 3 attempts to overcome all these shortcomings in the current implementation of FliX by enabling the load-awareness feature in FliX.
Chapter 3
Load-aware caching in FliX

In this Chapter we introduce the extended version of FliX. First, we give a formal
definition for the caching problem in FliX. Then we explore some well-known
caching algorithms together with their replacement strategies from the literature.
Finally, we present our caching algorithm and estimation function that we
implemented in FliX. In Chapter 4 we present detailed information about the
implementation of the load-aware caching in FliX and in Chapter 5 we present and
evaluate the results of the different experiments that we carried out to see how the
performance of query evaluation in FliX is improved.

3.1 Formal Model and Notation

3.1.1 Query Load
For a fixed corpus of linked XML documents, we consider descendant-or-self
path expressions like //book//author. We alternatively call such a path expression a
query \( q = (S,T) \) where \( T \) is a tag name and \( S \) is either a single element or a tag name.
In the former case we say that the query is single-source (SSQ), in the latter case all-
sources (ASQ). From now on we denote single elements with lower case letters (like
\( a,b,c, \ldots \)) and tag names with upper case letters (like \( A,B,C, \ldots \)).

The result of a query \( q = (S,T,d_{\text{max}}) \) is the set \( R(q) \) of tuples \( (s,b,d) \) where \( b \) is an
element with tag name \( T \) and \( b \) is a descendant of \( s \) with minimal distance \( d \leq d_{\text{max}} \).
The result of a query \( q = (S,T,d_{\text{max}}) \) is the set \( R(q) \) of tuples \( (a,b,d) \) where \( a \) is an
element with tag name \( S \), \( b \) is an element of tag name \( T \), and \( b \) is a descendant of \( a \)
with minimal distance \( d \leq d_{\text{max}} \).
Denoting the set of all possible queries by $Q$, a query load $QL$ of size $N$, which is a sequence of $N$ queries, can be formally defined as a map $QL: \{1 \ldots N\} \rightarrow Q$. When the query load $QL$ is clear from the context, we write $q_i$ instead of $QL(i)$.

For each query $q_i$, the set $R_i \subseteq R(q_i)$ is the subset of results of $q_i$ that the client actually reads. We consider a fixed query load $QL$ of size $N$. The absolute frequency $f(q_i)$ or short $f_i$ of a query $q_i$ is the number of times $q_i$ occurs in the query load,

i.e.,

$$f_i = | \{k: QL(k) = q_i\}|$$

The relative frequency $rf(q_i)$ or short $rf_i$ of a query is then defined as $rf_i = f_i / N$.

By $SC$ we denote the set of all sources of queries (ancestors) that appear in the query load, and by $f(s_j)$ and $f(S_j)$, we denote the number of times the node $s_j$, or the tag name $S_j$ respectively occurs in the query load as an ancestor in a path expression,

i.e.,

$$f_{s_j} = | \{ k : QL(k) = (s_j, T) \} |$$

$$f_{S_j} = | \{ k : QL(k) = (S_j, T) \} |$$

3.1.2 Evaluation Cost

We denote by $c(q)$ the cost to evaluate a single query $q = (s, T, d_{\text{max}})$ with FliX. This is the time needed to evaluate the query and can be computed as follows:

$$c(q) = c(s, T, d_{\text{max}}) = e(s, d_{\text{max}}) + r(s, T, d_{\text{max}})$$

where $e(s, d_{\text{max}})$ is the total cost of evaluating the tree of entry elements rooted at $s$. As described in Section 2.3, the depth of the tree of all entry elements reachable from source $s$ is a function of the maximum required distance of results $d_{\text{max}}$ that is set by the client at the beginning of the evaluation process. The second part in the right hand side of the equation above, $r(s, T, d_{\text{max}})$, represents the total cost of finding descendants of $s$ in all reachable meta-documents using local path indexes.
The cost to evaluate the whole query load $QL$ is then
\[
 c ( QL ) = \sum_{i=1}^{N} c ( q_i )
 = \sum_{s \in SC} f(s) \cdot e(s) + \sum_{q \in Q \setminus \{ QL \}} f(q) \cdot r(q)
\] (2)

To reduce the evaluation cost of the query load, we reserve some main memory with fixed size $M$ in which we cache the results of frequent queries. We also cache the tree of entry elements of frequent sources in the secondary memory. If we are able to answer a query $q$ from the cache, we define its total cost $c(q)$ to be zero. If we encounter a new query in the query load starting with a source node whose tree of entry elements is in the cache, we define the cost of evaluating entry elements $e(q)$ to be zero.

Given that queries are evaluated in FliX incrementally in several rounds, Equation (1) can be rewritten as follows:

\[
c(q) = \sum_{i=0}^{T_{\text{max}}} (e_{i}(s) + r_{i}(q))
\]

\[
c(q) = \sum_{i=0}^{T_{\text{max}}} e_{i}(s) + \sum_{i=0}^{T_{\text{max}}} r_{i}(q)
\] (3)

where:
- $T_{\text{max}}$ is the maximum number of rounds needed to retrieve all possible result nodes that are less than or equal $d_{\text{max}}$ steps away from the source.
- $e_{i}(s)$ is the cost of finding entry elements in round $i$
- $r_{i}(q)$ is the cost of evaluating descendants in round $i$

In general:
\[
 c(q) = \sum_{i=0}^{T_{\text{max}}} e_{i}(s) + \sum_{i=0}^{T_{\text{max}}} r_{i}(q) + \sum_{j=t+1}^{T_{\text{max}}} e_{j}(s) + \sum_{j=t+1}^{T_{\text{max}}} r_{j}(q)
\] (4)
Therefore, if the result nodes and entry elements of a previously evaluated query up to round \( t' \) were cached, the cost of evaluating query \( q \) that requires further evaluation steps to reach the required maximum distance of results can be defined as follows:

\[
c(q) = \sum_{j=1}^{T_{\text{max}}} e_j(s) + \sum_{j=t'+1}^{T_{\text{max}}} r_j(q)
\]

in some cases, where the complete entry elements tree of a previously evaluated query (that shares the same source \((s)\) as the new query \( q \)) is in the cache, the total cost \( c(p) \) can be computed as follows:

\[
c(q) = \sum_{j=t'+1}^{T_{\text{max}}} r_j(q)
\]

i.e., the cost of finding all descendants in steps \( t'+1 \) to \( T_{\text{max}} \) using local indexes, which can also be performed in one bulk step since the necessary entry elements are already given from the cache.

According to equations (2) and (4), it is obvious that the total cost of evaluating the query load \( QL \) can be reduced significantly if the result nodes and entry elements of frequent queries are cached.

Now, due to the space constraints in both main and secondary memory, our goal is to define caching strategies that minimize the evaluation cost using the cache. Depending on the amount of information the algorithm has, we get different approaches:

- In the offline approach, the algorithm gets as input the query load and the cost for each query. It then decides for each step of the evaluation if the result of that step is maintained in the cache, and if so, which entries are replaced in the cache if there is not enough room for the results. This algorithm produces the optimal result, but it usually cannot be applied in real systems.

- In the online approach, the algorithm incrementally sees the queries and has to decide at each query if it should be cached or not. This algorithm usually cannot compute the optimal solution, but can be applied in a real system.
In practice, search engines do not have the knowledge about the entire input, namely the query load, in advance. As such, our goal is to find and implement an online caching algorithm with a replacement strategy that insures that the limited dedicated resources (both main and secondary memory) are efficiently utilized.

3.1.3 Replacement Problem
We formally state the problem of the replacement policy in caching systems as follows: “given a limited cache capacity $M$, and a sequence $S$ of object requests, where each requested object $p$ in this sequence is associated with a retrieval cost, denoted by $cost(p)$ and a size, denoted by $size(p)$; the goal of the replacement policy is to decide on which object(s) in the cache to be removed to free up enough space for the newly retrieved object when the cache is full, i.e. there is not enough room in the cache for it”. In FliX, the objects to be cached are the resulting nodes of a query evaluation, namely, the descendants of a descendant-or-self path expression in addition to the entry elements that compose the so-called entry elements tree of the ancestor. Thus $cost(p)$ here corresponds to the evaluation cost $c(p)$ given by equation 1 above and $size(p)$ corresponds to the number of the result nodes or entry elements respectively. The sequence of objects $S$ in the problem statement above corresponds to the query load $QL$ in FliX.

For best performance we decided to apply the replacement policy in FliX in step-wise. This means the result nodes obtained from different steps of query evaluation are cached separately. Therefore when some result nodes are to be removed from the cache to allocate enough space for the newly evaluated query, the replacement policy is firstly applied on the result nodes of the last steps of the cached queries; i.e. the whole set of result nodes of a cached query is not removed entirely from the cache at once, rather, results of further steps are removed first, and if the freed up memory space is still not enough for caching the results of the new query, the replacement strategy is applied again on all cached queries that are yet left in the cache. By this way, we can ensure that some queries that reside in the cache and their size is too big are not replaced by some small queries just because there is no more free space in the
cache, instead, the removal of one or two steps of the cached query may suffice to provide the required space for the new query results.

### 3.2 Caching models

There are four models of caching:

1. **The Bit Model** [10]: In the Bit Model, for each object \( p \), we have \( \text{cost}(p) = \text{size}(p) \); i.e. the cost of evaluating or retrieving the object depends only upon its size.

2. **The Fault Mode** [10]: In the Fault Model, for each object \( p \) we have \( \text{cost}(p) = 1 \) while the size can be arbitrary.

3. **The Cost Model**: In the Cost Model, for each object \( p \), we have \( \text{size}(p) = 1 \) while the cost can be arbitrary.

4. **The General Model**: Finally, in the General Model, for each object \( p \), both the cost and size can be arbitrary.

Caching in FliX can be classified under the general model, since both of the cost (time) needed for evaluating the queries and the size (number of result nodes) is arbitrary.

### 3.3 Existing Replacement algorithms

We describe some cache replacement algorithms proposed from recent studies:

1. **Least-Recently-Used (LRU)**: this algorithm removes the object which has not been accessed for the longest period of time. This policy works well in workloads which exhibit strong temporal locality (i.e., recency of reference). LRU is a very simple policy requiring no parameterization. The frequencies of requests in the
recent history of the load does not play any role in this replacement policy, which is a good reason for us not to adopt this policy in FliX.

2. **Least-Frequently-Used (LFU):** this algorithm removes the object, which is least frequently requested. This gives objects that are frequently requested more weight than less frequent ones, which should work well for our purpose. However, applying this algorithm could lead to a cache setup where only a small number of very frequent but very large (in terms of size in bytes) objects being cached and objects that are a bit smaller but also a bit less frequent never get into the cache. An efficient replacement policy should attempt to cache as many objects as possible to increase the cache hits rate, i.e. to increase the number of requests that are answered from the cache.

3. **Size:** this algorithm removes the largest object from the cache but does not take the frequency of the object requests into account. Using this algorithm for our purpose may cause that some objects in the cache which are very costly (in terms of time of evaluation) are replaced by some objects which are little bit smaller in size but more costly. A situation which we would like to avoid in our cache.

4. **Lowest-Latency-First (LLF):** this algorithm removes the object with minimum cost; i.e. the object whose evaluation cost is the lowest. This algorithm proposes a solution for the problem of the latter algorithm (Size) but does cause new problems since it does not consider the frequency or the size of the object when taking the decision.

5. **LandLord:** is a generalization of the greedy-dual algorithm [11] for weighted Caching, which in turn generalizes least-recently-used and first-in-first-out replacement policies, as well as the balance algorithm for weighted caching. It was originally developed in the context of disk paging and, later on, was adapted to Web caching. This algorithm attempts to overcome all the problems that are not solved in the aforementioned replacement algorithms. In this algorithm, each
object in a cache set is associated with some credit. When an object is replaced, the credits of all objects remaining in the set are reduced by the current credit of the victim. Whenever an object is accessed, its credit is reset to its original cost. The object with lowest credit is always replaced. Figure 3.1 illustrates the different steps of LandLord algorithm. Initially, a newly cached object $p$ is assigned a credit which is proportional to its cost ($credit(p) = cost(p)$). Every time a new request to another object than $p$ occurs, the credit of $p$ is decreased. And every time a request to object $p$ is issued, its credit is reset to its start value, namely its cost. This way, the cost of evaluating object $p$ and both of its frequency and recency are important factors when taking the replacement decision. Unfortunately, LandLord algorithm does not work well when the cost differentials between objects are small, because the maximum credit an object gets is equal to its cost. Therefore, for two objects $p_1$ and $p_2$ where $p_1$ is quite more frequent than $p_2$, i.e. $f(p_1) >> f(p_2)$ but $c(p_1) < c(p_2)$, $p_1$ is faster removed from the cache than $p_2$ although its cost is a little bit smaller than $p_2$'s.

---

- Maintain a real value $credit[f]$ with each file $f$ in the cache.
- When a file $g$ is requested:

1. if $g$ is not in the cache then:
   2. while there is no room for $g$ in the cache:
      3. do
         4. For each file $f$ in the cache:
            5. do
               6. decrease $credit[f]$ by $\Delta \cdot size[f]$, where: $\Delta = \min_{f \in \text{cache}} \frac{credit[f]}{size[f]}$.
         2. od
      3. Evict from the cache any subset of the files $f$,
         4. such that: $credit[f] = 0$.
   5. od
   6. Bring $g$ into the cache and set $credit[g] = cost(g)$.
   7. else reset $credit[g]$ to CREDIT
   8. such that: current credit < CREDIT $\leq$ cost($g$).

---

**Figure 3.1**: LandLord algorithm for cache replacement policy
3.4 Caching in FliX

Finding an optimal replacement policy that fits for all application areas that are use online caching is almost infeasible. As such, we aim in this work to achieve an outstanding replacement policy that outperforms the well-known caching algorithms such like those that are described in Section 3.3.

3.4.1 Caching benefit estimation

We attempt to estimate the benefit of caching a new query result by exploiting the collected statistical information about the history of the query load. By means of such an estimation function a decision upon replacing an entry from the cache by a new one is made. Statistical information about the queries like the frequency of each query in the query load, how far is the last occurrence of each query in the recent past, the cost of evaluating each query and the size of the cacheable information of each one, are used to estimate the benefit of caching a given query. If the estimated benefit of caching a new query is larger than the benefit of keeping a cached query in the cache, the latter is replaced by the new one.

We define the caching benefit of query \( q \) as follows:

\[
\text{benefit}(q) = \frac{rf(q) \cdot c(q)}{a(q) \cdot |R(q)|}
\]

where:

- \( rf(q) \): relative frequency of \( q \)
- \( c(q) \): cost (i.e. evaluation time) of \( q \)
- \( |R(q)| \): number of result nodes of \( q \)
- \( a(q) \): age (i.e. how far is the last occurrence) of \( q \) in the query load

According to this benefit estimation function, frequent queries are given larger caching beneficial than infrequent queries. The parameter \( a(q) \) in the given estimation function represents the age of a given query with respect to the history of the query load. The age value varies between 1 and the current length of the query load \( N' \), such that, \( a (the \ most \ recently \ used \ query) = 1 \) and \( a (least \ recently \ used \ query) = N' \). Therefore, most recently used queries are assigned higher benefit than least recently
used ones. Higher priority for caching is given to queries with higher cost, this is important since our ultimate goal is to reduce the total cost of evaluating the whole query load. And due to the cache capacity constraints, we give queries with large size (number of results) less priority than small-size queries.

3.4.2 Caching optimization
In real applications, a client may stop the evaluation process of a given query at some point before the so-called Path Expression Evaluator has traversed the whole graph. Now in the same session, the client may refine the previously asked query by increasing or decreasing the required maximum distance specified in the last action. From the system point of view, this means that the same query has occurred multiple times, but with different requested maximum distance for results. As discussed in Section 2.3, depending on the requested maximum distance (dmax), the query evaluation process takes a variant number of rounds in FliX. Thus, the different versions of the query share a minimum number of evaluation rounds. Therefore, to optimize our caching algorithm in FliX, we propose to implement a round-level caching mechanism.

It is likely the case in some real applications that a large number of the ancestor-descendant queries share the same originating source (i.e. ancestor). For instance, queries like \texttt{author//*} that share the same ancestor \texttt{‘author’} are very frequently used to retrieve data out of a digital library like DBLP. Based on this discussion and to further optimize the caching mechanism in FliX, we also aim to enable reusing the cached information to answer new queries. This can be achieved by caching the common entry-elements trees of frequent sources.

3.4.2.1 Caching in round-level
As each round in the query evaluation process has its own sources and results, it can be considered as a sub-query. For further optimization we decided to cache the result nodes of each round separately. By collecting information about each round of the
query evaluation of each query in the query load, the benefit estimation function that is proposed in Section 3.4.1 can be easily applied to decide upon caching the result nodes and/or the intermediate-sources at the end of each round. This way, we can avoid replacing a complete set of results of some query from the cache by very small but very frequent queries. Instead, a small set of the results that belong to the estimated minimal beneficial round of the query can be replaced. Consider the two queries $q_1$ and $q_2$ shown in Figure 3.2 as an example. Query $q_1$ is a cached query, which was evaluated in 5 rounds. Each round is numbered and cached separately. The blocks in Figure 3.2 represent the cached results of each round. Both the size and the cost of each block are arbitrary and are given. Query $q_2$ is a newly evaluated query, which consists of only 2 rounds.

$$
\begin{array}{cccc}
1 & 2 & 3 & 4 & 5 \\
\text{s=5} & \text{s=10} & \text{s=6} & \text{s=3} & \text{s=15} \\
\text{c=10} & \text{c=15} & \text{c=8} & \text{c=6} & \text{c=20} \\
\end{array}
$$

$q_1$: in the cache

$$
\begin{array}{cc}
1 & 2 \\
\text{s=3} & \text{s=4} \\
\text{c=10} & \text{c=10} \\
\end{array}
$$

$q_2$: new

**Figure 3.2:** replacement policy in the round-level

Assume that the estimated benefit of keeping $q_1$ in the cache is less than the estimated benefit of caching $q_2$ and the free space in the dedicated memory is less than the required space for caching $q_2$; a naïve approach would suggest to remove the whole query $q_1$ from the cache and to insert $q_2$ into the cache. In our proposed replacement strategy, we do compare the benefit of caching each block separately. In the context of our example, we firstly compare the benefit of caching the first block of $q_2$ with the benefit of keeping the last block of $q_1$. If block1 of $q_2$ is more beneficial than block5 of $q_1$, we remove block5 of $q_1$ from the cache. Since the size of block5 is larger than

37
of block1 and block2 of \( q_2 \) together, query \( q_2 \) can now easily be inserted into the cache. Thus, \( q_1 \) need not to be entirely removed from the cache.

Another benefit for keeping statistical information from the query load in the level of evaluation rounds is the capability of answering new queries that share the same query-prefix of some cached queries; i.e. queries that are similar to some previously evaluated queries but with larger requested distance for the results. As an example, a query \( q_1 = (s, T, 5) \) requires result nodes with tag name ‘\( T \)’ that are descendants of the source node ‘\( s \)’ and are at most 5 steps far a way from ‘\( s \)’. A new query \( q_2 = (s, T, 8) \) is similar to \( q_1 \) with the exception that the maximum distance for the results is 8. If \( q_1 \) is in the cache and intermediate sources of its last round are cached, we can directly return the result nodes of \( q_1 \) that are up to distance 5 and use the intermediate-sources of the last round to continue evaluating the query until the new set maximum distance is reached. This saves a significant amount of time hence reducing the total cost of evaluating \( q_2 \).

### 3.4.2.2 Caching entry-elements tree

We have shown in Section 2.3 that evaluating the entry-elements during the query evaluation is time consuming. Hence, caching such tree of entry-elements a long with meta information like the identifier of the meta-document it belongs to and the distance between the originating source (ancestor) and this element, will allow us to answer new queries that share the same source quickly in one step by determining the descendant meta-documents that hold the cached entry-elements and using their local indexes in parallel in one big bulk operation. As discussed in Section 3.1.2, the total cost of the query load can be computed by the following equation:

\[
c(QL) = \sum_{s \in SC} f(s) \ast e(s) + \sum_{q \in Q(QL)} f(q) \ast r(q)
\]

where the frequency \( f(s) \) is usually greater than one. By caching the entry-elements tree of frequent sources, the term \( f(s) \) in the above equation can be set to 1, hence reducing the total cost of the query load.
The resulting nodes can easily be ranked in terms of the distance between each result node and the originating source by simply adding the distance between the result node and its local ancestor (entry element) to the distance between its local ancestor and the source of the query.

3.4.3 Caching Algorithm

Here we introduce the proposed caching algorithm. In Figure 3.3-(a) we show an overview of the query evaluation procedure after enabling caching in FliX. As an input, FliX takes a query \( q = (\text{source}//\text{target}) \) as an input. In real applications, a client may stop the query evaluation process at any point when (s)he is satisfied with the early returned results. As the Path Expression Evaluator in FliX returns the retrieved results to the client ordered in a distance-based ascending order, in our formal model we consider the maximum distance \( d_{\text{max}} \) of the subset of results that the client actually reads as an input to our system. At the end of each query evaluation round, FliX returns the results of that round directly to the client over a pipeline and checks if the client has stopped the evaluation process or not; i.e. if \( d_{\text{max}} \) has been reached.

As shown in Figure 3.3-(a), FliX checks if the results of the given query \( q = (\text{source}//\text{target}, d_{\text{max}}) \) are already in the cache; if yes, they will be returned directly to the client, otherwise the query is sent to the PEE (Path Expression Evaluator) to execute one query evaluation round. The results of the executed round are returned to the client and if \( d_{\text{max}} \) is reached the process will stop; otherwise a new evaluation round will start. FliX continues evaluating the given query until either \( d_{\text{max}} \) is reached or there are no more rounds to run; i.e. no more meta-documents to traverse.

At the beginning of each evaluation round, PEE checks whether the intermediate-sources (i.e. entry elements) for this round are in the cache. If the required intermediate-sources are found in the cache, PEE starts using them to retrieve new results for the given query in the corresponding meta-documents.
However, if PEE does not find the required intermediate-sources in the cache, it must first compute them and then start using them to retrieve the results.

After each round of the query evaluation, new results or intermediate-sources that are evaluated using the PEE are sent to the cache. Based on the replacement policy and the benefit estimation function that we discussed in section 3.4.1, a decision upon caching the new entries is made. Figure 3.3-(b) illustrates the algorithm of the cache management in FliX.

The algorithm takes the set of results (or intermediate-sources) of the new round as an input, together with the collected statistical information from the history of the query load, and attempts to insert the new entries into the cache. If the size of the new results (sources) is less than or equal to the free space in the cache, the algorithm inserts these results (sources) immediately into the cache. If there is not enough space in the cache for the new entries, the algorithm has to make a decision on removing some entries from the cache to free up enough memory for the new ones. First, the benefit of caching the new results (sources) is estimated. Second, the benefit of keeping each round of each query in the cache is estimated. To achieve this, statistical information about the frequency, recency, cost and size of each round are used. After that, the algorithm compares the benefit of caching the results (sources) of the new round with the minimum benefit of all entries in the cache. If the benefit of the new round is smaller, nothing is being cached, otherwise the set of results (sources) of the round with the minimum benefit is removed from the cache, and the algorithm is repeats itself until enough space in the cache are allocated for the new entries.
Figure 3.3-(a)
Figure 3.3-(b)
Chapter 4
The Implementation

In this chapter we describe our implementation of the load-aware caching in FliX. We have extended the current implementation of FliX that was introduced in [2]. The programming language that we used is the Sun Java-VM version 1.4.2 on Microsoft Windows XP Professional operating system and Oracle 9.2 is used for the database.

4.1 Extended Architecture
Figure 4.1 shows the extended architecture of FliX. Statistical information about the workload and each particular query in the recent history of the query load are maintained by FliX to help making a replacement decision by the cache when it is needed. Regarding the query load itself, information like when and how frequent each query in the recent history has occurred are collected. Detailed information about each query the maximum distance of the result elements that the client actually reads, the size of the set of results of each round of the query evaluation process, the cost (time) of the evaluation in each round are also gathered. In the Cache of FliX the results of each query are cached separately from the entry-elements tree of its source. Therefore, the cache in FliX consists of two blocks: one for caching the result elements of frequent queries and the other for caching the entry-elements trees of frequent sources (ancestors). Statistical information for each block is maintained separately; i.e. information like the frequency, recency, size and cost of each ancestor is also maintained for the purpose of estimating the benefit of caching a new entry-elements tree.

Because of the great impact of caching entry-elements trees on the performance of the query evaluation, and due to the memory constraints, we decided to cache only the results of frequent queries in the main memory and to cache the entry-elements trees of frequent ancestors in the form of tables in the database. As such, the cache of the entry element trees is extensible, meaning its maximum capacity can be increased.
easily and with a relatively low cost. The replacement algorithm which uses the benefit estimation function that we introduced in Section 3.4.1 is applied for both fragments.

**Figure 4.1**: Extended FliX Architecture

### 4.2 Cache Structure

For the purpose of collecting statistical information about the query load, we consider a fixed-size window of the recent history of the query load. The size of the window is configurable and can be adjusted according to the available memory resources.

We implemented and appended a new class called FliXCACHE that is responsible for managing the cache in FliX as well as maintaining statistical information about the query load. We also modified the different functions of PEE of FliX such that it can communicate with the cache for exchanging the results and/or the entry elements of each round. For instance, at the beginning of each evaluation round, the PEE has to
ask the cache for any cached entry elements for the corresponding round to speed up the evaluation process. Also at the end of each evaluation round, the PEE should send the newly retrieved results to the cache. In the rest of this Section we describe our implementation of the two blocks of the cache in FliX, namely, the cache of results and the cache of entry-element trees.

4.2.1 Caching results

We summarize our implementation of this fragment of the Cache as follows:

- A class called CachedQuery is implemented to represent each entry of the cache
- We maintain a List $CQ$ of the cached queries (objects of the type CachedQuery).
- Each CachedQuery instance maintains detailed information about each of its evaluation rounds
- A CachedQuery object maintains a hash table that maps each round number to its set of results $RoundResults$ in addition to the evaluation cost of that round $RoundCost$
- We use an ArrayList for the purpose of maintaining statistical information about the query load window $QLW$.
- The size of $QLW$ is configurable.
- Any query occurs in the query load is inserted at the beginning of the this list $QLW$
- A long with each query in the list $QLW$, the following meta-information are kept:
  - the maximum distance of the results that the client actually read ($d_{max}$)
  - the number of the last evaluation round ($lastRound$) that was executed to evaluate this query
- For the purpose of estimating the benefit of caching a given evaluation round $r$ of a given query $q$ the estimation function that we introduced in Section 3.4.1 is used:

$$benefit(r) = \frac{rf(r) \times c(r)}{a(r) \times |R(r)|}$$
In the following we describe how the different parameters, that are needed to estimate the caching benefit of a given query \( q \), are computed:

- The frequency of the query \( rf(q) \) : count the occurrences of \( q \) in query load window \( QLW \)
- The frequency of a single round \( r \) of query \( q \) : compute \( rf(q) \) such that \( q.lastRound \geq r \)
- The age of the query \( a(q) \): the smallest index \( i \) of \( QLW \) such that \( QLW(i)=q \)
- The age of a single round \( r \) : compute \( a(q) \) such that \( q.lastRound \geq r \)
- The size (number of results) of a single round \( r \), \( |R(r)|=size(RoundResults) \)
- The cost of a single round \( r \), \( c(r) = RoundCost \)

### 4.2.2 Caching Entry Elements Trees

We summarize our implementation of this fragment of the Cache as follows:

- We maintain one table \( EETable \) of all cached entry-elements trees
- Each cached entry-element is stored in one record in the \( EETable \)
- Each record (entry-element) in the \( EETable \) has the following fields:
  - \( EntryElementID \): the ID of this entry-element
  - \( OriginAncestor \): the origin ancestor of this entry-element, i.e., the root of the entry-elements tree that this entry-element belongs to.
  - \( MetaDocumentID \): the ID of the meta-document that contains this entry element
  - \( Distance \): the distance between this entry-element and the origin ancestor
  - \( Round \): the number of the evaluation round , to which this entry-element belongs
- Given a query \( q = (source, target) \), FliXCache can retrieve all the corresponding entry-elements of the ancestor \( source \) in one simple SQL select statement:

  ```sql
  Select * from EETable where OriginAncestor = source
  ```
And by using of the MetaDocumentID of each entry-element the corresponding local path index is determined and used to evaluate local descendants of those entry-elements, which in turn are descendants of the origin ancestor source. By adding the Distance value of each entry-element to the distance value of each of its own local descendants, FliX can rank all resulting nodes in an ascending distance-based order with respect to the origin ancestor.

- We maintain a table for representing the query load window QLWT
- Each query occurs in the query load is stored in one record in QLWT
- Each record in the QLWT has the following fields:
  - OriginAncestor: the origin ancestor of the query
  - LastRound: the number of the last evaluation round that was needed to evaluate this query
  - Timestamp: to represent the time of occurrence of this query in the query load

- A table CostTable for maintaining meta-information about each evaluation round for each query in the query load is used
- Each record in the CostTable stores information about one evaluation round
- Each record of the CostTable has the following fields:
  - OriginAncestor: the origin ancestor of this entry-element, i.e., the root of the entry-elements tree that this entry-element belongs to.
  - Round: the number of the evaluation round
  - Cost: the cost of this evaluation round

- For the purpose of estimating the benefit of caching the entry-elements of a given evaluation round r of a given query q = (source, target) the estimation function that we introduced in Section 3.4.1 is also used here:

\[
\text{benefit}(r) = \frac{rf(r) \cdot c(r)}{a(r) \cdot |R(r)|}
\]
In the following we describe how the different parameters, that are needed to estimate the caching benefit of a given entry-elements tree, are computed:

- The frequency of a single round \( rf(r) \):
  \[
  \text{count} \ast \text{from } QLWTable \\
  \text{where } \text{OriginAncestor}=source \text{ and lastRound } \geq r
  \]

- The age of a single round \( a(r) \):
  Order the table QLWTTable in ascending order by Timestamp and count all records that are ranked above the first record that has the same OriginAncestor value and its lastRound value is greater than or equal \( r \).

- The size (number of entry-elements) of a single round, \( |R(r)| \):
  \[
  \text{count} \ast \text{from } EETable \text{ where} \\
  \text{OriginAncestor}=source \text{ and Round}= r
  \]

- The cost of a single round, \( c(r) \):
  \[
  \text{Select Cost from CostTable} \\
  \text{where } \text{OriginAncestor} = source \text{ and Round}= r
  \]
Chapter 5

Experimental Results

In this Chapter we experimentally evaluate the effectiveness of our load-aware caching system in FliX. The results that we present here show the significant improvement in the performance of FliX in the context of evaluating a collection of queries.

5.1 Setup

All our experiments were run on a Windows–based PC with a 3GHz Pentium 4 processor, 1 GByte RAM and 120 GB Disk space. We used an Oracle 9.2 database that ran on the local disk.

We used the first implemented version of FliX which is introduced in [2] for building the Framework (meta-documents, their path indexes and FliXLinks) for the subset of DBLP [15] that consists of 6210 XML documents of size 13.2 MB. Table 6.1 shows the statistics of the XML data model of the subset of DBLP.

<table>
<thead>
<tr>
<th>Docs</th>
<th>Nodes</th>
<th>Edges</th>
<th>Links</th>
<th>Inter-links</th>
<th>Intra-links</th>
</tr>
</thead>
<tbody>
<tr>
<td>6210</td>
<td>168991</td>
<td>162781</td>
<td>25666</td>
<td>25684</td>
<td>18</td>
</tr>
</tbody>
</table>

**Table 5.1:** XML collection model

To build the query load for our experiments, we created 50 different queries randomly, 29 query of the type $X/Y$ and 21 query of the type $x/Y$. Then we built the query load manually by repeating some queries to give them different frequencies in the query load. We also inserted some queries into the query load several but with
different $d_{\text{max}}$ value (the maximum distance of the results). There is no single query in the query load being requested more than 8 times with the same $d_{\text{max}}$ value.

### 5.2 The Replacement policy

We compare the replacement policy of our caching algorithm, which is based on the caching benefit estimation function that we introduced in Section 3.4.1, with the replacement policy of LandLord caching algorithm. For this experiment we built a query load of 100 queries as described in Section 5.1. We compared the performance of each algorithm in with respect to the total evaluation cost of the whole query load and the hit rate of each one. By the hit rate we mean, the ratio of the number of requests, which were answered directly from the cache to the total number of requests in the query load. Figures 5.1 and 5.2 show that caching in FliX outperforms the LandLord caching algorithm in both factors.

In Figures 5.4-5.6 we show the difference between the replacement policies in FliX’s caching algorithm and LandLord’s caching algorithm with respect to two sample queries Query1 and Query2. In our experiment, Query1 and Query2 are equal in terms of the evaluation cost and the size of the results set. Query1 is very frequent within the first 40 requests, after that it does not show up in the query load, whereas Query2 is quite frequent between the request number 15 and the request number 45 in the load, after that its frequency increases strongly until the end of the 100 requests. Figure 5.3 shows how frequent does each of the sample queries occur in the query load.

By using LandLord’s replacement policy, both Query1 and Query2 receive the same amount of the Credit (which corresponds to the caching benefit value in our caching algorithm) during the first phase of the load, although Query1 is more frequent in this phase than Query2. In the second phase of the query load, we see that Query2 loses its Credit although it is very frequent in this phase. This is because both queries have the same cost, and according to LandLord algorithm, the maximum
Credit that a query in the cache can receive is equal to its cost. Hence, any increase in the frequency of the query does not reflect into its Credit value. Another reason for the bad performance of the LandLord algorithm in this context is that it does not give any priority to the most recently used queries, thus, these queries loose their Credit values very quickly.

On the other hand, using our caching algorithm, the caching benefit of Query1 and Query2 does efficiently reflect their frequency and recency as shown in Figures 5.6 and 5.7.

![Comparison between Replacement Policy of FliXCach and LandLord](image)

**Figure 5.1:** Comparison between the total evaluation cost of the whole query load in FliXCach and LandLord cache
Comparison between Replacement Policy of FliXCache and LandLord

Figure 5.2: Comparison between the cache hit rate in FliXCache and LandLord cache

The frequency of two sample queries in the query load

Figure 5.3: The sample queries Query1 and Query2 in the query load.
Figure 5.4: The caching credit of sample query Query1 using LandLord algorithm

Figure 5.5: The caching credit of sample query Query2 using LandLord
Figure 5.6: The caching benefit of sample query Query1 using caching algorithm in FliX

Figure 5.7: The caching benefit of sample query Query2 using caching algorithm in FliX
5.3 Caching Entry-Elements Trees

We also ran some experiments to show the significant impact of caching and reusing the common entry-elements trees on the performance of FliX in the context of query evaluation. For this purpose, we do not cache any results, rather, we cache only entry-elements (intermediate-sources). Figure 5.8 shows the experimental results of two different runs, the first run, where we turned the cache off and the second run where we cache only intermediate-sources. The evaluation cost (time in ms) of each request in the query load is shown for both cases. The results of this experiment shows that the total evaluation time of the query load can be significantly reduced in general, and for some cases, the algorithm can save more than 60% of the cost. For this experiment we used a large query load of size 175.

![Evaluation cost of different queries in the query load](image)

**Figure 5.8:** Evaluation cost of different queries in the query load
5.4 The Performance of the Extended FliX

In Figure 5.9 we compare the overall performance of the Path Expression Evaluator in FliX in the context of evaluating a large collection of queries in two different settings:

1- without implementing any caching algorithm
2- with the capability of both caching results and entry-elements and reusing common entry-elements trees to evaluate new queries

We evaluated the same query load (175 queries) with different cache capacities. For the second scenario, we fixed the capacity of the entry-elements cache to 50000 entry-elements.

According to the given chart in Figure 5.9, the larger the cache capacity is, the smaller the total evaluation cost is getting. The results show that caching in FliX has improved the performance of query evaluation significantly.

![Figure 5.9: The total evaluation cost of the whole query load](image-url)
Chapter 6
Summary and Outlook

6.1 Summary

Summarizing the presented work, this thesis extends the FliX framework, for indexing connections in large collection of inter–linked XML documents, by integrating a load-aware caching mechanism into its first implementation in [2].

We discussed the concepts of path expression evaluation in FliX and pointed out the shortcomings of the current implementation in the context of evaluating a large collection of queries.

In this thesis, we presented our caching replacement policy and its benefit estimation function which exploits collected statistical information about the history of the query load, to estimate the benefit of caching a new query.

Different form the traditional page-based caching systems, caching in FliX exploits the idea of reusing cached information to answer new queries. Our experimental results reveal that our load-aware caching in FliX has significantly reduced the total cost of evaluating a collection query load with a factor of about 4.
6.2 Outlook

Our future work aim to the following regards:

- We will carry out additional experiments with larger query loads and larger sets of XML documents.

- We want to further examine the performance of our caching algorithm by fixing the size of the cache and evaluating several different and ‘complete’ randomly generated query loads.

- We plan to enhance the load-awareness in FliX by exploiting statistical information about the workload in the build phase. For instance, if it turns out in the query evaluation engine that most queries have to follow many links over some time, then the current setup of the meta-documents is no longer optimal. In that case, the build phase should start again, taking in account the collected statistics over the query load.
Literatures


[11] Neal E. Young. The k-server dual and loose competitiveness for paging. Algorithmica,


