Incremental Reclustering of Augmented XML Trees

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Shnaiderman Lila

Sumbitted to the Senate of the Technion - Israel Institute of Technology

Heshvan, 5769 Haifa November, 2008
This Research Thesis Was Done Under the Supervision of Professor Oded Shmueli in the department of Computer Science.

I would like to thank Prof. Oded Shmueli for his invaluable support and guidance throughout this research.
I would also like to thank my husband, Arie, to which this work is dedicated, for his love, support and understanding.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>4</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>1.1 Objectives</td>
<td>11</td>
</tr>
<tr>
<td>1.2 Related Work</td>
<td>13</td>
</tr>
<tr>
<td>1.3 Thesis Organization</td>
<td>19</td>
</tr>
<tr>
<td>2 The System</td>
<td>20</td>
</tr>
<tr>
<td>2.1 Simulated Disk and Disk Page Layout</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Cache Management</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Saxon-Like XPath Processing (Saxon-L)</td>
<td>23</td>
</tr>
<tr>
<td>3 The Incremental Reclustering Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>3.1 The Temporal Proximity Algorithm (TPA)</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Incremental Reclustering</td>
<td>34</td>
</tr>
<tr>
<td>3.3 Incremental Triggering</td>
<td>36</td>
</tr>
<tr>
<td>4 Experimental Evaluation (Simulated Disk)</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Main Parameters</td>
<td>43</td>
</tr>
<tr>
<td>4.2 Experiments Description</td>
<td>44</td>
</tr>
<tr>
<td>4.2.1 Setting Up</td>
<td>44</td>
</tr>
<tr>
<td>4.2.2 Experiment Description</td>
<td>45</td>
</tr>
<tr>
<td>4.3 Evaluation of PIXSAR</td>
<td>46</td>
</tr>
</tbody>
</table>
5 Experiments with a Real (Physical) Disk

5.1 Main Parameters ............................................. 57
5.2 Results of Experiments ..................................... 59

6 iPIXSAR ................................................................ 65

6.1 Index-based Incremental Reclustering ....................... 67
6.2 Example ............................................................... 70
6.3 Evaluation of iPIXSAR ......................................... 74

7 System Architectures ............................................. 80

7.1 Using PIXSAR in a single disk context ....................... 80
7.2 Extensions for Novel Architectures ......................... 83
  7.2.1 Two Levels Mass Memory ................................. 83
  7.2.2 Web site ......................................................... 89

8 Conclusions and Future Work ................................ 92

Bibliography ................................................................ 99

A Lukes’ Algorithm ................................................. 100

B XML Generator’s DTD .......................................... 103

Hebrew Abstract ....................................................... a
List of Figures

1.1 A two cluster partition of an augmented tree. .................. 8

2.1 Page layout scheme. Slots: 1-Beginning of free space; 2-Number of slots in the directory; 3-Pointer to an overflow page; 4-Initial global counter index; 5-Initial inner counter; 6-Initial outer counter; 7-Incremental inner counter; 8-Incremental outer counter. ........ 22

2.2 Saxon log example; the query is /a/b/c and the path index is on /a/b. ................................................. 27

3.1 TPA. ................................................................. 33
3.2 The deleteFromHead operation. ................................. 34
3.3 The IncrementalR Algorithm. .................................. 37
3.4 Reclustering Algorithm. ........................................ 41

4.1 Comparing page faults for document with size 10MB. The x axis is the number of queries executed in units of 40,000/300. 48

4.2 Comparing page faults for document with size 124MB. The x axis is the number of queries executed in units of 40,000/300. 49

5.1 Comparing DFS with PIXSAR for document with size 10MB. The x axis is the number of queries executed in units of 600,000/1000. ................................................................. 62

6.1 The IdxIncrementalR Algorithm. ............................... 68
6.2 Augmented tree types. ............................................ 70
6.3 Function runIncrRecluster. ...................................... 71
6.4 Index Reclustering Algorithm. .................................. 73
6.5 Comparing page faults for a document of size 10MB. The $x$ axis is the number of queries executed in units of 40,000/300. The $y$ axis is the improvement of iPIXSAR in comparison to PIXSAR.

7.1 Architecture schemes. (a) Self-Reclustering Disk (b) Clustering Operating System (c) Clustering Database.

7.2 Two levels memory example.

7.3 Cut edge weights example.

7.4 The Two Memory Levels Algorithm

7.5 The Webiste Storing Algorithm

A.1 An XML Tree partitioned by using Lukes’ Algorithm.
List of Tables

2.1 Page Directory .................................................. 28
2.2 Node Data Unit ................................................... 29

4.1 Results for a 10MB document. In all the experiments, the number of DFS page faults is 11,466,303, the DFS cut value is 13,483,580. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. ............... 50

4.2 Results for a 124MB document. In all the experiments, the number of DFS page faults is 1.28E+08, the DFS cut value is 1.42E+08. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. ............... 51

4.3 Results for a 256MB document. In all the experiments, the number of DFS faults is 2.45E+08 , the DFS cut value is 2.77E+08. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. ............... 52

4.4 Results for indexed queries. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. ................................................................. 54

4.5 Results for the "changing workload" experiment. The Impr columns show percentage improvement of the first run in comparison to the second run. ................................. 55

5.1 Results for a 10MB document. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. In all the experiments, the number of DFS page faults is 57,400,167. ......................................................... 60

5.2 Results for a 10MB document. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. ......................................................... 61
5.3 Results for documents with other sizes. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. The DFS run in the first experiment was performed for 100,000 queries only and for 160,000 in second experiment, so the numbers in the table that are marked with '*' were extrapolated which is justified as the DFS line is always straight.

5.4 Time improvement, showing percentage improvement of an incremental run in comparison to a DFS run.

6.1 Results for a 10MB document. PF columns show number of page faults. The Impr1 column show percentage improvement of a PIXSAR run in comparison to a DFS run. The Impr2 column show percentage improvement of an iPIXSAR run in comparison to a PIXSAR run.

6.2 Results for a 124MB document. The Impr columns show percentage improvement of an iPIXSAR run in comparison to a PIXSAR run.
Abstract

XML is one of the primary encoding schemes for data and knowledge. We investigate incremental physical data clustering in systems that store XML documents using a native format. We formulate the XML clustering problem as an augmented (with sibling edges) tree partitioning problem and propose the PIXSAR (Practical Incremental XML Sibling Augmented Reclustering) algorithm for incrementally clustering XML documents. We show the fundamental importance of workload-driven dynamically rearranging storage. We also touch upon extending PIXSAR for handling insertions and deletions. PIXSAR incrementally executes reclustering operations on selected subgraphs of the global augmented document tree. The subgraphs are implied by significant changes in the workload. As the workload changes, PIXSAR incrementally adjusts the XML data layout so as to better fit the workload. PIXSAR’s main parameters are the radius, in pages, of the augmented portion to be reclustered and the way reclustering is triggered.

We also adapted PIXSAR to operate on a physical disk. Further we suggest novel architectures that utilize PIXSAR in multi-level storage. We propose
a specific algorithm for one of these architectures.

We use an experimental data clustering system that includes a disk simulator and a File System simulator for storing native XML data. We mainly consider static files but also explain how dynamically updated files can be handled. We use a novel method for 'exporting' the Saxon query processor into our setting. Experimental results indicate that using PIXSAR significantly reduces the number of page faults (counting ALL page faults incurred while querying the document as well as those during maintenance operations) thereby resulting in improved query performance (often by 20%-40%). We also conducted extensive experiments on a physical disk. Interestingly, the improvements observed on the real disk are much larger (often by more than 50%) than those predicted by simulations.

In addition to the XML augmented tree, there are also indices. The kind of index we consider is based on a XPath expression and it consists of index entries pointing to XML target nodes. Using such index entries one jumps directly to target nodes. Often, target XML nodes are accessed in temporal proximity and hence, for paging reasons, it is beneficial to store them on the same disk page. In other cases, such temporal proximity is absent and hence co-storing is not optimal. Designing an algorithm that views the XML data and indices as a sibling augmented tree with multiple roots (the additional roots correspond to indices) is complex. An extension to the PIXSAR algorithm, called iPIXSAR, is proposed. It extends PIXSAR so as to make storing decisions of target XML nodes based on possible membership in more
than one tree. Experimental results show that in the presence of indices iP-IXSAR is superior to PIXSAR by at times up to 8%.
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>Cache Size</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>DS</td>
<td>Document Size</td>
</tr>
<tr>
<td>DT</td>
<td>Disturbance Time</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>EIC</td>
<td>External Incremental Counter</td>
</tr>
<tr>
<td>EPC</td>
<td>External Preliminary Counter</td>
</tr>
<tr>
<td>EUF</td>
<td>Edge Update Frequency</td>
</tr>
<tr>
<td>GC</td>
<td>Global Counter</td>
</tr>
<tr>
<td>IPC</td>
<td>Internal Preliminary Counter</td>
</tr>
<tr>
<td>IIC</td>
<td>Internal Incremental Counter</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>MRT</td>
<td>Multi Rooted Tree</td>
</tr>
<tr>
<td>OODB</td>
<td>Object Oriented Data Base</td>
</tr>
<tr>
<td>PIXSAR</td>
<td>Primitive Incremental XML Augmented Clustering</td>
</tr>
<tr>
<td>RF</td>
<td>Reclustering Factor</td>
</tr>
<tr>
<td>RPL</td>
<td>Reclustering Page Limit</td>
</tr>
<tr>
<td>QN</td>
<td>Queries Number</td>
</tr>
<tr>
<td>RR</td>
<td>Reclustering Radius</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TPA</td>
<td>Time Proximity Algorithm</td>
</tr>
<tr>
<td>XML</td>
<td>Extensive Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

XML [30] is in widespread use for both data and knowledge encoding. XML is a platform independent standard whose hierarchical nature is useful in modeling the real world. There are numerous works on languages and systems for efficiently accessing XML data [14, 6, 24, 1, 7]. This work deals with storing XML data on secondary storage. The way storage is organized dramatically affects performance. We show that a static, workload independent, storage scheme is vastly inferior to one that incrementally adjusts storage layout. We show this for a particular static storage scheme, DFS, which is used for initial data storage. However, the phenomenon we observe are highly likely to hold for any static storage scheme. For large data sets, it is completely impractical to frequently perform total data rearrangement and the practical choices are either keeping a static initial layout or incrementally adjusting the data layout. Many systems have evolving access patterns. For example,
if an actor receives the Oscar, access to her past movies in a movies database are likely to increase significantly. So, the problem we address and the methods we present are fundamental for any large scale XML data or knowledge system and to principally-hierarchical systems in general.

There are two main approaches for storing XML documents. The first approach maps an XML document to a relational table where each row represents an edge in the document’s XML tree [7]. Relational operators are used for traversing over XML stored documents. The second approach, native XML Storage, views an XML document as a tree. The entire XML tree is partitioned into distinct records containing disjoint connected sub-trees [14, 6, 24, 1]. These records are stored on disk pages, either in an unparsed, textual form, or using some internal representation.

In a relational database the disk organization essentially follows the relations’ schemes. Partitioning of data to pages is the result of a data loading algorithms and the disk structures employed; for example, a loading algorithm for a $B^+$ tree or a hash table. Keys are the main proximity factor affecting physical closeness of records.

In native XML Database systems, navigation over the document is strongly influenced by the document hierarchical structure. The main navigational tool is XPath (which can also use indexes). So, physical clustering of navigationally related data items is beneficial. Here, two document nodes are related if they are structurally connected via an edge, and examining one of them is likely to soon lead to examining the other. In past work, data
clustering has been shown to be beneficial for hierarchical databases [22], and for object-oriented databases (OODBs) [5, 29, 10, 21].

The workload, namely which queries and in what frequency and importance level, determines which data is frequently accessed. Hence, it is important for the physical organization to match the workload. However, the workload may change, which means that ideally the data physical organization need change as well. To give solution to this problem, we defined a new algorithm called PIXSAR (Practical Incremental XML Sibling Augmented Reclustering). PIXSAR is the only algorithm we know of that incrementally adjusts physical XML data placement to match a changing workload. PIXSAR can also be extended to the case in which the file content changes as well. We also extended our methodologies to handling remote storage.

We formulate the XML clustering problem as an augmented (with sibling edges) tree partitioning problem. The tree to be partitioned is a clustering tree, namely an augmented XML tree with node and edge weights. Roughly, the edge weights model the (usually XPath) navigational behavior (higher edge weights mean that the connected XML nodes are traversed more often in temporal proximity). Node weights measure the size of node data in bytes including: the text size of the XML node as well as various references (parent, sibling, etc.) and weights associated with parent and sibling references (edges). The problem PIXSAR addresses, is to partition the set of nodes of the clustering tree into node-disjoint subsets (clusters) so that each cluster fits into a disk page, and the total of the intra-cluster edges’ weights (called
the partition’s value) is maximized. Intuitively, a higher value partition results in fewer disk accesses. Figure 1.1 displays a partition of an augmented tree consisting of two clusters.

In addition to the XML augmented tree, there are also indices. Database systems use path indices that reduce the number of navigation steps across stored XML nodes. Thus, disk-resident XPath processors employ a mixed, i.e., part navigational, part indexed, processing model. The kind of index we consider is based on a XPath expression and it consists of index entries pointing to XML target nodes. Using such index entries one jumps directly to target nodes. Often, target XML nodes are accessed in temporal proximity and hence, for paging reasons, it is beneficial to store them on the same disk page. In other cases, such temporal proximity is absent and hence co-storing is not optimal.

At first, XML clustering was explored in the context of the basic tree (Lukes
algorithm [20], XC algorithm [2], primary NATIX algorithms [6]). Later on, clustering was extended to a tree augmented with sibling edges (XS [4, 3] NATIXs new algorithms [15]). In this thesis we address a new problem that, as far as we know, was never investigated, clustering in the context of a Multi Rooted Tree (MRT). Practically, we consider each index as an additional root to some subtrees of the basic tree. For an n-node tree there is no known $O(nW^k)$ precise partitioning algorithm for this problem (where $W$ is the page size limit and $k$ is a constant); this problem is non-trivial and is left for future work.

We extend PIXSAR to iPIXSAR. iPIXSAR uses a heuristic method, based on comparing the partitions that are induced independently by the trees in the MRT. iPIXSAR exploits the observed XML navigational behavior, through structural edges as well as indices, to direct its incremental partitioning activity. The basic idea is that it considers (imaginary and weighted) edges between adjacent index target nodes. Incremental clustering modifications are based on both the PIXSAR algorithm as well as on weights assigned to these imaginary edges. The assignment of weights to these imaginary edges is based on an adaptation and extension of PIXSARs edge weight assigner. The intuition underlying iPIXSAR is as follows. iPIXSAR selects trees out of the full document graph. The trees include the one corresponding the basic XML hierarchical structure as well as trees implied by indices. It determine the overall ”profit” in the value of the partition that PIXSAR can obtain based on each of these trees. Then reclustering for only the most ”profitable
tree” is performed.

Our work is not limited to traditional XML documents. For example, we consider very large hierarchical documents that encode whole web sites. Viewed as a whole, the web site may be regarded as essentially tree-based, with relatively few additional links that ”break” the tree structure. So, we extended PIXSAR and wrote an algorithm that can handle these tree-like structures. We also consider documents that encode a large database that needs two mass storage levels. These two mass storage levels may be realized locally (on slow devices) or even remotely (say, via a service such as Amazon S3 [26]). Managing data in such an environment raises intricate optimization problems that involve both time and money (the remote service may charge for both storage and operations). We propose an algorithm that extends PIXSAR to this setting.

To evaluate our ideas we have constructed an extensive infrastructure. It includes a simulated, memory resident, disk and a simple File System to manage it (because it is impractical to do extensive experimentation on an actual disk). To allow workload tracking and data reclustering, a detailed disk-page format was designed and implemented. Another issue was how to generate the query workload. The method used is to take an industrial strength XPath query processor, Saxon [16], trace its operations on actual XML files and transform the navigational behavior to our setting. On this infrastructure, extensive experimentation on a large combination of relevant parameters was conducted.
After getting great results on a simulated disk, we also experimented with a real (physical) disk with the following characteristics: capacity - 500 GB, rotational speed: 7,200 rpm, cache - 16 MB, interface - SATA II, max. external transfer rate - 300 MB/s, seek time read - 8.9 ms, seek time write: 10.9 ms, seek time track-to-track - 2.0 ms. We showed that PIXSAR in a physical disk environment is much better than predicted by simulations.

1.1 Objectives

The initial objective of this work was to design algorithms for incremental reclustering of augmented XML trees. This led to the PIXSAR algorithm and to the PIXSAR system. Some of the main contributions are as follows:

- The main contribution of this work is PIXSAR, a practical algorithm for incremental workload-directed XML reclustering. PIXSAR views the XML file as an augmented tree (with sibling edges). Edge weights encode temporal access proximity. PIXSAR exploits the navigational behavior to direct its incremental partitioning activity. The main characteristics of PIXSAR are as follows:
  - It makes decisions on the fly, and selectively reclusters parts of the augmented document tree that experience significant changes in access behavior.
  - PIXSAR is tunable - the main parameters are the radius, i.e., the portion to be reclustered, and the sensitivity of reclustering
- PIXSAR uses the temporal proximity algorithm (TPA) which, while traversing the document, incrementally computes the co-temporal access affinity between structurally related document nodes.

- PIXSAR may be used in conjunction with any edge-weight-based data partitioning algorithm and an initial data placement produced by any algorithm.

- An experimental data clustering system that includes a fast disk and File System simulator for storing native XML databases, and extensive experimentation of the PIXSAR algorithm within this system.

- A novel method for simulating an existing query processor in a new environment. We introduce the Saxon-L query processor which mimics the Saxon query processor in our environment.

- Extensive experimentation of PIXSAR over a real disk.

- We present in detail the Temporal Proximity Algorithm which is the "heart" of the PIXSAR algorithm. It records cache residency temporal proximity via edge weights.

- iPIXSAR, an extension of PIXSAR for incremental XML reclustering of documents with XPath indices. iPIXSAR views the XML file as a sibling-augmented tree in which some of the nodes are target nodes of indices (i.e., pointed by index entries). This gives rise to a multi-rooted
tree (MRT). Extensive experimentation with iPIXSAR is presented.

- We explore architectural options for embedding PIXSAR within a disk-based system.
- We examine possible PIXSAR extensions. We consider a two levels mass storage system in which the second level is handled by a remote storage service. We propose an algorithm for managing data movements between the two levels. We also consider managing a whole web site via PIXSAR and propose an algorithm for this task.

1.2 Related Work

Tree Partitioning Problem

Consider a rooted tree $T = (V,E)$, where $V$ is a set of nodes and $E \subseteq V \times V$ is a set of edges. A cluster over $T$ is a non-empty subset of $V$. When no confusion arises, we simply use the term cluster. A partition of $T$, $P_T$, is a set of pair-wise disjoint clusters over $T$ whose union equals $V$, that is $P_T = \{c_1,\ldots,c_k\}$, $k \geq 1$, such that $\bigcup_{i=1}^k c_i = V$, and $c_i \cap c_j = \emptyset$, for all $i \neq j$. Each node $i$ of $T$ is associated with a non-negative integer weight, $w_i$. Each edge $(i,j)$, of $T$ is associated with a non-negative integer value, $v_{ij}$. The size of a cluster $c$, $size(c)$, is the sum of the values of its nodes; formally, $size(c) = \sum_{i \in c} w_i$. The value of a cluster $c$, $value(c)$, is the sum of the values of its edges; formally, $value(c) = \sum_{(i,j) \in E \land i \in c \land j \in c} v_{ij}$. The value of a partition $P_T$, $value(P_T)$, is the sum of the values of its clusters; formally,
\( \text{value}(P_T) = \sum_{c \in P_T} \text{value}(c). \)

Let \( W \), the \textit{cluster weight bound}, be a positive integer. The \textit{tree partitioning problem} is formulated thus. Find a highest value partition, \( P_{\text{opt}}^T \), among all the possible partitions of \( T \), such that the size of each cluster in \( P_{\text{opt}}^T \) does not exceed \( W \). \( P_{\text{opt}}^T \) is said to be an \textit{optimal partition} of \( T \) (in general there may be more than one optimal partition). So \( P_{\text{opt}}^T = \{c_1, ..., c_k\} \) such that \( \text{size}(c_i) \leq W \), for \( i = 1, ..., k \), and \( \text{value}(P_{\text{opt}}^T) = \text{Max}\{\text{value}(P^T)|P^T \text{ is a partition of } T \text{ and } \forall c \in P^T, \text{size}(c) \leq W\} \).

Determining whether there exists a tree partition with a value of at least \( v \) is NP-complete in the ordinary sense (Problem ND15, acyclic partitioning [9]). Lukes [20] presents a linear time algorithm for tree partitioning that incorporates edge weights. It finds a partitioning of optimal value, e.g., one where the total weight of all edges that do not go across partitions is maximized. For unit edge weights, the algorithm solves the same problem as the Kundu and Misra algorithm (discussed below). It finds a partition with minimal cardinality. Lukes’ does not consider sibling partitioning.

Johnson and Niemi propose two algorithms for this problem [12]. Both exhibit higher running times than Lukes’ algorithm. In [8], linear-time algorithms are presented for partitioning trees with edge weights so as to minimize (maximize) the weight of the largest (smallest) cluster by removing \( k \) edges.

Kundu and Misra [19] proposed an \( O(n) \) tree partitioning algorithm (\( n \) is the number of nodes). Unfortunately, the \( O(n) \) running time is possible only for unit edge weights and the algorithm requires the entire tree to be in memory.
Partitioning a graph based on inter node affinity and frequency of access, appears in other contexts [13, 17, 18], including for example Bond-Energy based program text partitioning [11].

**Physical Clustering of Related Data**

Since the early days of database technology development, physical clustering of related data items has been examined for exploiting data access patterns. The problem was addressed also in hierarchical databases and in the context of OODBs [22, 29, 10].

An early reported application of clustering for physical database design was in hierarchical databases [22]. Object-oriented databases (OODBs) have used clustering of logically related objects for achieving better physical object placement [29, 10]. Object clustering in OODBs has not been widely accepted primarily due to the complexities of object-oriented programming (e.g., frequently changing object access patterns) and problems with effectively partitioning the resultant object graph. OODBs traditionally use graph partitioning algorithms for partitioning a clustering graph whose node weights represent object sizes and edge weights denote access behavior. Such algorithms exhibit large space and time requirements. Furthermore, these graph partitioning techniques are not suitable for partitioning trees as they do not exploit structural aspects of trees for making partitioning decisions. Also, these algorithms assume the entire graph to be generated before partitioning and need *multiple in-memory* passes over the generated graph. Interestingly, [10] uses Lukes’ tree partitioning algorithm when the clustering graph is a
tree and concludes that it is not useful in real applications due to its large memory usage.

Schkolnick [22] considers an organization in which data segments are arranged in a hierarchical manner (i.e., as a tree) and accessed via transitions between segments, beginning at the root segment (referred to as hierarchical scan). He considers a probabilistic model for paging behavior and addresses the issue of clustering the data segments so that page faults are minimized. He does not view the clustering problem as a tree partitioning problem and focuses on only those paths that originate from the tree root. Object clustering can be sequence-based, in which a graph encoding object relationships (object graph) is transformed into a sequence of objects which is then sequentially mapped to pages [29]. An alternative scheme is partition-based clustering where the object graph is partitioned into clusters of related objects, where the cluster size is bound by the underlying page size.

Gerlhof et al. [10] examine clustering in object-oriented databases using the graph partitioning approach. Their goal is to place together logically related objects (in equal sized pages). Their partitioning algorithm uses a clustering graph with node weights representing object sizes and edge weights denoting access behavior. They propose a greedy graph partitioning heuristic and compare it against other algorithms for clustering objects. A related problem, that of modeling database access patterns for clustering objects has also been examined. A key work on this topic is the stochastic modeling of object clustering by Tsangaris and Naughton [28]. They propose two mod-
els for exploiting the access information, namely, the Independent Identical Distribution (IID) model and the simple Markov Chain Model, and examine the clustering problem for these two models. Similar approaches could be applied to modeling XML traversals as well.

**Clustering XML Documents**

Zhang et al. proposed a physical storage scheme that clusters XML nodes related by parent-child and following-sibling/preceding-sibling relationships [31]. Their clustering approach neither uses the tree partitioning formulation nor uses workload information.

In [6], Fiebig et al. describe Natix, a pioneering native XML system. In Natix, XML documents are split into multiple subtrees (records) such that each subtree record can fit into a page. Recently, Kanne and Moerkotte have developed a linear-time XML clustering algorithm for the Natix system [15]. Their algorithm uses sibling relationships between nodes and its goal is to minimize the number of clusters in the partition. They also proposed additional, more practical, approximate algorithms. They showed experimentally the importance of taking sibling relationships into account while clustering, they experimented with files whose sizes are up to 11MB. The limitations of the Kanne-Moerkotte approach, which are addressed by this thesis, are as follows:

- It takes node weights (size) into account but *does not* consider edge weights in the augmented XML tree.
• It does not exploit workload information during the clustering process.

• It does not address the issue of reclustering, incremental or otherwise, as the document access pattern changes over time.

• There is no way to incrementally record the workload information as the document access patterns change over time.

A practical approximate algorithm, called XC, that clusters whole XML documents using a tree partitioning approach, is presented in [2]. XC uses XML (which usually means XPath) navigational behavior, which is recorded as parent-child edge weights, to direct its document partitioning. XC is based on Lukes’ tree partitioning algorithm [20] (see also Appendix A), but in contrast to Lukes’ algorithm, which is an exact algorithm, XC is an approximate algorithm. XC trades off partitioning precision for time and space. This enables XC to exhibit linear-time (in document size) behavior without significant degradation in partitioning quality over the exact optimal solution.

However, performing clustering based on navigational behavior as encoded in parent-child edge weights is not sufficient. It misses the fact that often children of a parent are accessed successively. This means that to reduce the number of page faults, affinity among sibling nodes should also be taken into account. XS [4, 3], an extended version of the XC algorithm, clusters an XML document taking into account navigational affinity among sibling nodes. XS, like XC, is a practical approximate algorithm that trades off precision for time and space. XS can effectively partition an XML document whose static
workload characteristics are known.

The algorithms mentioned thus far (XC, XS and the algorithm of Kanne and Moerkotte [15]) use static workload information while partitioning data. In contrast, PIXSAR incrementally adjusts, and improves, the storage layout according to a dynamically changing workload. This allows PIXSAR to continuously improve the data placement on disk to match the continuously changing workload.

1.3 Thesis Organization

This thesis is organized as follows:

- Chapter 2 provides an overview of the experimental platform.
- Chapter 3 presents PIXSAR.
- Chapter 4 describes the experiments with PIXSAR.
- Chapter 5 describes experiments with a physical disk.
- Chapter 6 presents the iPIXSAR algorithm.
- Chapter 7 gives an overview of possible PIXSAR extensions.
- Chapter 8 concludes and suggests future research directions.
Chapter 2

The System

We present an experimental platform for evaluating the PIXSAR algorithm. The platform is composed of a simulated or physical disk for storing XML documents, a special disk page layout, a simple cache management system and the Saxon-L simulator which mimics the Saxon XPath query processor. Note that the PIXSAR algorithm operates within a system that manages its own page cache independently of the Operating System (OS). In experiments with a physical disk, cache management is implemented via completely bypassing the OS by using the Unix raw disk option.

2.1 Simulated Disk and Disk Page Layout

We implemented a simple, in-memory, disk simulator which stores and manages XML tree nodes. The disk is implemented as a linear array of pages
Each page contains a page directory and page data, as illustrated in Figure 2.1. The page directory holds the meta information for the page. The relevant information is described in Table 2.1. Page data is composed of node data units that are described in detail in Table 2.2. It is important to notice that the fields that describes sibling edges are optional. I.e., left sibling pointer, right sibling pointer, left sibling weight and right sibling weight fields exists only when a node has left or/and right edges. For example all our algorithm do not use right sibling edges, so right sibling pointer and right sibling weight fields are never used. The following terms are used in the table:

**Page internal weight**: The sum of the weights of all edges with both ends located in the page.

**Page external weight**: The sum of the weights of all edges from nodes in the current page to nodes in other pages.

**Discussion**: The reason for simulating the disk was to speed up experimentation with the PIXSAR algorithm. For that aim, we keep the simulated disk data in main memory (to expedite the experiments). We abstract out the physical characteristics of the disk (such as arm, platters, rotation, disk cache, etc.) by viewing it as an array of pages. While we lose precision using this simplification, we expected that the ranking of results will not significantly change based on a more detailed simulation or in a physical (real) disk. The experimentation that we have performed with physical (real) disk has shown that this is indeed the case. A similar approach to predict relative
performance lies at the foundations of query processing, such as system R (Griffiths et al. [25]).

### 2.2 Cache Management

Disk data is read into a memory resident cache. This cache manages and traces disk accesses during run time. We use LRU as the page replacement discipline. As required by the reclustering triggering algorithm, a page may be pinned in main memory. Consequently, we use LRU with the following
modifications. The page that is removed, in case that a frame for a new page is needed, is the least recently used unpinned page. The removed page is written to disk only in case that it has been changed since the time it was read into memory. We use the LRU page replacement discipline for ease of experimentation. In fact, any other replacement discipline could have been used in our internal cache management. The particular discipline that is used does not influence the incremental reclustering or the edge updating algorithms; i.e., these algorithms are orthogonal to the used discipline. As the system we are envisioning bypasses the operating system and is therefore not affected by whatever page replacement algorithm used by the operating system.

### 2.3 Saxon-Like XPath Processing (Saxon-L)

The Saxon-L (Saxon Like) simulator mimics the operations of the Saxon XPath query processor [16]. The main difference between Saxon and Saxon-L is that Saxon stores XML information in relational tables, while Saxon-L uses native XML Storage. Saxon navigates the XML document using the following information:

1. The "next" column of the table ("next" contains the following sibling node and, in case there are no following sibling nodes, "next" returns the parent node).

2. IDs of the nodes which are processed in DFS (depth first search) order.
3. The "parent" column of the relational table.

Saxon-L is a hypothetical query processor which operates in a manner similar to that of Saxon. A log, of nodes that are visited during the execution of the Saxon XPath processor, is created. We "execute" Saxon-L by (hypothetically) using the Saxon traversal algorithm, and mimicking the traversal carried out by Saxon. This is possible as both Saxon and Saxon-L traverse each axis exactly in the same way. Consider, for example, traversing a child axis. Saxon performs it by using the next column. It obtains the first child (the next node in DFS order) and then iterates over all the following sibling nodes. Saxon-L simply goes over all the children pointers that are located within the node record. Consider traversing a descendant axis. Saxon considers, in DFS order, all the nodes that have an ID bigger than the ID of the current node and smaller than the ID of the following sibling of the current node. Saxon-L iterates over all the descendants of the current node, in DFS order, by using a recursive function that implements a simple DFS traversal of the augmented tree. This is why the log file, which tracks each time a node in the augmented tree is touched by Saxon, records an identical traversal order for both tools, namely the actual Saxon and the hypothetical Saxon-L. In other words, following the recorded execution in the Saxon log tantamount to the order the hypothetical Saxon-L would have produced.

During processing a query, many passes are made over children of a node, the node itself and its ancestors. So in order to reduce number of pages while processing a query, Saxon-L is required to place all the ancestors of the
current node in an Ancestors Stack and remain there during the time that it processes the current node. Saxon-L does it by pinning any node that is placed in the Ancestors Stack in main memory. This method reduces the number of page faults as it guarantees that if some node is in main memory, its ancestors are also in main memory. Saxon-L operates in page mode (rather than node mode) which means that during the time that some node of a page is in the Ancestors Stack, the entire page remains in main memory. This hinges on the assumption that the tree depth of XML documents is usually small (a few tens at most).

**Supporting indexes:** Indexes contain direct pointers to a set of nodes that correspond to some XPath query expression. When a query expression whose prefix is an indexed expression is processed, no traversal is done for the "indexed" part of the expression - the query processor "jumps" directly to the nodes that are pointed by the index. Consequently, no weight is added to edges which are included in the path that is covered by the index. The (hypothetical) Saxon-L query processor (as opposed to Saxon) utilizes indexes in XPath queries. We implement this feature by inserting a simple modification into the Saxon source code that enables marking the indexed part of the query execution. During the processing of the query, the modified Saxon puts a special mark, in the produced log, next to the nodes that are part of the navigation covered by the index. Saxon-L identifies this mark, and does not read the corresponding nodes (and hence no page faults are caused). This way we simulate the direct jump, by Saxon-L, to indexed
Discussion: The weight updating algorithm (see section 3.1) determines whether nodes are accessed in temporal proximity, in which case storing them on the same disk page is likely to reduce the number page faults. Note that in case temporal proximity applies, but nodes are connected by neither child-parent edge nor by sibling edge, we are unable to record this property and perhaps more intricate mechanisms are needed. This occurs, for example, when using indexes.

Implementation details: Saxon-L is implemented using the following steps:

1. Create the log of Saxon’s run for the given XPath queries that form the workload.
2. In the Saxon-L simulator, traverse the augmented document tree by using the log produced in the previous step, and update (during the traversal) the weight of the edges which are not covered by an indexed part, as described above.

Recall that Saxon-L holds an Ancestors Stack of the nodes on path between the root of the document and the currently visited node. Each time that a new node is traversed, the Ancestors Stack and the corresponding edge weight are updated.

Figure 2.2 presents a simple example of a Saxon-L run. In the example we see a simple augmented document tree, and a log of the query /a/b’/c for this tree. The character ‘ denotes the last step of an indexed part, i.e., an
Figure 2.2: Saxon log example; the query is /a/b/c and the path index is on /a/b.

index exists for /a/b. The first column of the Saxon log is the DFS ID of the touched node. The second column (that contains '∗' characters) indicates whether the current node is part of a path covered by an index. The order of traversal can be determined from the log, i.e., the DFS ID of the first node that is touched is ”-1” (it is a dummy node that symbolizes the root of the augmented tree - parent of a), the next touched node has DFS id 0 (the id of the real root of the document - the node a), etc. Note that all the nodes that belong to the indexed part /a/b are marked with ∗. When Saxon-L traverses the document according to this log, it does not read ’∗’ marked nodes from disk. That is, no weight is added to edges (−1, 0), (0, 1), (0, 3) and (0, 4), because the nodes with DFS ID 0, 1, 3 and 4 are covered by the index expression /a/b. The weights of edges (1, 2), (4, 5) are incremented during the traversal.
<table>
<thead>
<tr>
<th><strong>Slot Name</strong></th>
<th><strong>Slot Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to empty place</td>
<td>Indicates the beginning of the free space of this page.</td>
</tr>
<tr>
<td>Number of entries in the directory</td>
<td>Indicates the number of nodes stored in the current page.</td>
</tr>
<tr>
<td>Overflow Pointer</td>
<td>In case that some document node is bigger than a page, the overflow pointer points to a page which is a continuation of the current page.</td>
</tr>
<tr>
<td>Pointer to initial global counter</td>
<td>This counter holds the sum of the weights of all the augmented tree edges at the time of the last reclustering.</td>
</tr>
<tr>
<td>Initial internal counter</td>
<td>Holds the sum of the weights of internal page edges at the time of the last reclustering.</td>
</tr>
<tr>
<td>Initial external counter</td>
<td>Holds the sum of the weights of external page edges at the time of page creation.</td>
</tr>
<tr>
<td>Incremental internal counter</td>
<td>Holds the sum of weights that were added to the internal page edges during traversals over the page that have been performed since the last reclustering.</td>
</tr>
<tr>
<td>Incremental external counter</td>
<td>Similar to incremental internal counter.</td>
</tr>
<tr>
<td>Node Info Entries</td>
<td>Every node in the page has an entry in this page directory. It contains: the node offset which indicates the start address of the node data, node data length in bytes, a dirty bit that indicates whether the node info entry can be deleted from the page directory in case that the node is deleted, a left sibling bit that indicates if a left sibling pointer exists, and a right sibling bit.</td>
</tr>
</tbody>
</table>

Table 2.1: Page Directory.
<table>
<thead>
<tr>
<th>Field name</th>
<th>Field description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Pointer</td>
<td>A pointer to the parent node of the current node.</td>
</tr>
<tr>
<td>Edge weight</td>
<td>Weight of the edge between the parent node and the current node.</td>
</tr>
<tr>
<td>Left sibling pointer</td>
<td>Optional field. In case a left sibling edge exists, this is a pointer to the left sibling node.</td>
</tr>
<tr>
<td>Left sibling weight</td>
<td>Optional field. In case a left sibling edge exists, this is the weight between the left sibling node and the current node.</td>
</tr>
<tr>
<td>Right sibling pointer</td>
<td>Similar to left sibling pointer.</td>
</tr>
<tr>
<td>Right sibling weight</td>
<td>Similar to left sibling weight.</td>
</tr>
<tr>
<td>Attribute counter</td>
<td>Number of attributes of the current node. Note that every attribute is held in a different node.</td>
</tr>
<tr>
<td>Children nodes counter</td>
<td>Number of children of the current node.</td>
</tr>
<tr>
<td>Attribute pointers</td>
<td>List of pointers to attribute nodes.</td>
</tr>
<tr>
<td>Children pointers</td>
<td>List of pointers to children nodes.</td>
</tr>
</tbody>
</table>

Table 2.2: Node Data Unit.
Chapter 3

The Incremental Reclustering Algorithm

This chapter describes the core algorithms of the PIXSAR system.

PIXSAR incrementally improves document placement on disk based on the actual querying activity. As time progresses, the workload associated with the document may change. The main idea behind reclustering is to adjust the document placement on disk (after an initial clustering) to the changing workload, so that the number of page faults is reduced. The trivial solution is to perform a full reclustering each time the workload is changed. But, this is inefficient as it requires reading and clustering the entire document, and these are very complex and extremely slow operations. PIXSAR opts for incremental reclustering. PIXSAR detects areas that are either “underweight” or “overweight” with respect to the rest of the document and recluster them.
3.1 The Temporal Proximity Algorithm (TPA)

For simplicity we assume queries are evaluated one at a time and that recluster-
ing may only be performed on query boundary (that is, between executing
successive queries) and not while processing a query. During the traversal
over the document tree, edge weights are updated. An edge weight quanti-
fies the importance of placing its two nodes on the same page on disk. The
idea is that if two nodes are temporally traversed ”close enough” to each
other, they should be located in the same disk page. The meaning of ”close
enough” is as follows. Suppose a node $x_1$ is traversed and then, after travers-
ing nodes $x_2, x_3$ to $x_{i-1}$, node $x_i$ is traversed. In case that nodes $x_1, x_2 \ldots
x_i$ are small enough to be placed in the same disk page, $x_1$ and $x_i$ are ”close
enough”. If during the traversal two nodes are found to be ”close enough”
and there is an edge between them, either parent-child or sibling, this edge’s
weight is incremented by one. As will be further explained in section 4, the
augmented tree that is stored in disk is built in such a way that an edge is
directed and is stored only at its source node. The nodes of an edge $(a,b)$
may be encountered ”close enough” in both orders, first $a$ then $b$ and vice
versa. This creates some complications. The temporal proximity algorithm
copes with these complications; the idea is to note a future necessary update
that will possibly be executed if the ”other” node of some edge is visited
”soon enough”.

We use two data structures in this algorithm. The first is a queue $Q$, which
contains the last visited nodes in FIFO order so that the sum of the sizes of all nodes in $Q$ is less than page size. The second is a hash table $H$ whose keys are nodes. A key (node) $u$ has two associated pieces of data in $H$. The first is a flag that states if $u$ is in $Q$ (inQueue flag). The second is an edges-list which contains edges, whose weights may need to be increased in the near future, such that $u$ is one of the edge’s nodes. We also use two functions. The first, $p(u)$, returns the parent of node $u$, and if $u$ has no parent, it returns null. The second, $ls(u)$, returns the left sibling of the node $u$ and if $u$ has no left sibling, it returns null.

TPA has three parts. The first part manages the queue. It adds the currently traversed node $u$ to $Q$ and fixes $Q$ so that the sum of the sizes of all nodes in $Q$ would be less than page size. The second part updates the weight of the corresponding edges in case that $u$ is in $H$ (as it means that both nodes of these edges are in the $Q$ at this moment), otherwise adds $u$ to $H$. The third part updates the weight of the parent and left sibling edges of $u$ in case that they are currently in the $Q$, otherwise it stores these edges in $H$ (as maybe we will meet these edges soon and then we would update them).

TPA is presented in Figure 3.1. The $deleteFromHead$ operation is described in Figure 3.2. Its purpose is to delete all the data of the currently deleted node from the algorithm’s data structures.

TPA has one limitation. It is suitable only for trees that all their nodes are smaller than a page size.

Another important thing is that TPA works in conjunction with any XPath
For node $u$:

1. if ($u$ is in $Q$)
   1.0 move it to the tail of $Q$
   else /*$u$ is not in $Q$*/
   1.1 while ($(Q$.totalWeight + $u$.weight) > pageSize)
   1.2 $Q$.deleteFromHead /*(described later on)*/
   1.3 add $u$ as the tail of $Q$

2. if ($u$ is in $H$)
   2.0 $u$.inQueue = on
   2.1 for each $e$ in $u$.edge_list
   2.2 update the weight of $e$
   2.3 delete $e$ from $u$.edge_list
   else /*$u$ is not in $H$*/
   2.4 add $u$ to $H$
   2.5 $u$.inQueue = on

3. if ($p(u)! = null$ and $p(u)$ is in $H$)
   3.0 if ($p(u)$.inQueue = on)
   3.1 update the weight of $(u, p(u))$
   3.2 else
   3.3 add $(u, p(u))$ to $p(u)$.edge_list
   else /*$p(u)$ is not in $H$*/
   3.4 add $p(u)$ to $H$
   3.5 add $(u, p(u))$ to $p(u)$.edge_list

4. Do as in step 3, but for $ls(u)$ instead of $p(u)$

Figure 3.1: TPA.
for node \( v \):

/*update \( H \):*/
1. if (\( v.\text{edgeList} \) is not empty)
   1.0 \( v.\text{inQueue} = \text{off} \)
   else
   1.1 delete \( v \) from \( H \)

/*update other data:*/
2. if (\( p(v) \) is in \( H \) and (\( v, p(v) \)) is in \( p(v.\text{edgeList}) \))
   2.0 delete (\( v, p(v) \)) from \( p(v.\text{edgeList}) \)
   2.1 if (\( p(v.\text{inQueue} = \text{on} \) and \( p(v.\text{edgeList} \) is empty)
   2.2 delete \( p(v) \) from \( H \)
3. Do as in step 2, but for \( ls(u) \) instead of \( p(u) \)
4. Delete \( v \) from \( Q \)

Figure 3.2: The \textit{deleteFromHead} operation.

execution algorithm (and not only for the one that is used by SAXON). The reason is that TPA operations depend only on the order in which the document nodes are visited; it is independent of the particular navigation algorithm that produced the order.

### 3.2 Incremental Reclustering

The \textit{Reclustering Radius} determines the pages to be reclustered. Intuitively, the radius reflects the maximum distance of pages that are affected by a change in a page.

More formally, it is defined inductively:
• Base case: \( R_0 \), the set of pages that correspond to a 0 reclustering radius, contains only the page on which a significant change occurred.

• Inductive step: let \( R_{i-1} \) be the set of pages that correspond to a reclustering radius of \( i - 1 \). \( R_i \) is defined as \( R_{i-1} \cup \{ \text{page} \mid \text{page contains a node directly connected via an edge to a node located in a page which belongs to } R_{i-1} \} \).

The decision as to whether reclustering is required is based on changes in pages rather than nodes. This is due to space considerations. To this end, each page contains special data items for determining whether reclustering is needed. These special items include initial and incremental counters of internal and external edge weights associated with the page and the index of the global counter which contains the sum of all edge weights in the entire augmented tree at some point in time. The global counter is part of an array of global counters. Each slot in this array stores the index and the size of the corresponding counter. Following each reclustering, the current global counter is added to that array (i.e., the counter gets a new index and then the index and the size of the counter are stored in the array). In case that there is no free slot in the global counter array, we conduct a cleaning operation on the array. During this operation, we remove each third global counter. In case that the algorithm needs to access a deleted global counter, it uses the global counter with the closest index to the deleted one. The traversal algorithm is described below.

Incremental reclustering algorithm (\( \text{IncrementalR} \)) uses \( \text{XS} \) procedure to per-
form reclustering operations. This procedure takes as input a data subtree (augmented with sibling edges) and produces a partitioning of the subtree into a set of clusters. XS is based on the augmented tree clustering algorithm of [4, 3] and on the techniques used in the clustering systems of [2] which mainly deal with limiting memory usage and processing time. XS produces an approximate solution whose accuracy depends on the available memory and time resources.

We use two variables in this algorithm. The first is \( rFlag \) (reclustering flag), which is set as soon as we find that reclustering is needed (it is initially set to \textit{off}). The second is \( rNode \) (reclustering node), which is set to hold a pointer to the node that caused the reclustering decision. The \texttt{random()} function returns a random integer (0 to 100); \( q \) is a system parameter (0 to 99).

The algorithm is presented in Figure 3.3. The motivation of the "if" in line 4 is to reduce the number of times in which we check whether reclustering is needed. The function \texttt{toRecluster} in line 4.0 is explained in section 3.3. The motivation of line 4.1 is to provide an indication of the necessity of a reclustering in line 1.0.

### 3.3 Incremental Triggering

While traversing the augmented tree, edge weights are modified as described in Section 3.1. Changes of some edge weights may lower the quality of the partition, and reclustering of some part of the augmented tree is useful. The
For each traversed node $v$:

1. if ($v$ is first node of the query) /*just beginning a new query*/
   1.0 if ($rFlag = on$)
   1.1 $rFlag = off$
   1.2 The part to recluster, $RP$, consists the page in which $rNode$ is located and all pages within the reclustering radius.
   1.3 Convert $RP$ into procedure XS’ internal data format, $IRP$.
   1.4 Run XS on $IRP$.
   1.5 Assign each resulting cluster to a page having sufficient space to hold it. During each assignment do as follows:
      1.5.0 Write the current state of the page (internal and external) weights into the page counters.
      1.5.1 Add the current global counter to the global counters array.
      1.5.1 Write the ID of the current global counter to the page.

2. Update the weights of all edges that connect $v$ to other nodes that are "close enough" by using the TPA algorithm.

3. Update the incremental weight counters of $v$’s page.

4. if ($random() < q$).
   4.0 if ($toRecluster(v)$)
   4.1 if $rFlag = off$
   4.2 $rFlag = on$
   4.3 $rNode = v$

Figure 3.3: The IncrementalR Algorithm.
triggering algorithm determines when a change of an edge weight should cause reclustering.

An *internal* (respectively, *external*) edge is an edge connecting two nodes residing in the same (respectively, different) disk page (respectively, pages). Intuitively, an edge should cause reclustering when its weight is changed significantly, relatively to all other tree edges. As, it means that traffic through this edge has significantly increased/decreased relative to other edges. When an edge is an external edge, and the traffic through it is smaller, the value of the partition is also relatively improved. In a similar way, when an edge is an internal edge, and the traffic through it becomes larger, the partition value is relatively improved. So, reclustering need be triggered only when an edge is an external edge and the traffic through it relatively increases, or when an edge is an internal edge and the traffic through it relatively decreases. The triggering of reclustering is based on changes in *pages* rather than nodes. It is simply impractical to track changes at the level of nodes. To formally describe the triggering test, we define the following counters: \(IIC\) - Internal Incremental Counter, \(EIC\) - External Incremental Counter, \(IPC\) - Internal Preliminary Counter, \(EPC\) - External Preliminary Counter, \(GC_{OLD}\) - Global Counter (the sum of all edge weights in the document) in some moment \(t\) right after previous reclustering of this page, \(GC_{NEW}\) - Global Counter at the moment of running the test, in time \(t+t_1\). and \(TRF\) - Triggering Reclustering Factor (how much change has to occur).

The boolean function \(toRecluster\) determines if reclustering is needed. It
return true when one of the following holds:

- The percent of $IIC + IPC$ out of the new $GC$ becomes smaller by at least $TRF$ in comparison to the percent of $IPC$ out of the old $GC$, i.e.,
  \[
  \frac{(IIC + IPC)}{GC_{NEW}} / \frac{(IPC/GC_{OLD})} \leq \frac{(100 - TRF)}{100}.
  \]

- The percent of $EIC + EPC$ out of the new $GC$ becomes bigger by at least $TRF$ in comparison to the percent of $EPC$ out of the old $GC$, i.e.,
  \[
  \frac{(EIC + EPC)}{GC_{NEW}} / \frac{(EPC/GC_{OLD})} \geq \frac{(100 + TRF)}{100}.
  \]

To illustrate the PIXSAR algorithm, we present an example in Figure 3.4. For this example, we define the page size to be 3 (say KB), node size to have a fixed size of 1, $TRF$ to be 2% and $Radius$ as 1. The left side of Figure 3.4(a) shows the disk state after running initial reclustering, and before starting traversing the document. We observe that the $IPC$ and $EPC$ counters contain the corresponding initial values, while $IIC$ and $EIC$ are zero, because the traversal has not yet started. The right side of Figure 3.4(a) shows the disk state after many traversing operations, right after traversing edge ($a, f$) i.e., while visiting node $f$. $IIC$ and $EIC$ are updated and contain the difference between the current and the initial values of the edge weights. Next, we run the triggering test on page 1 (the page in which node $f$ is placed) and discover that the percent of $IPC$ out of $GC_{OLD}$ is 30% (13/44) while $IIC + IPC$ out of $GC_{NEW}$ is 27% (23/85). Since $27/30 = 0.90 < 0.98$, triggering occurs. We conclude that reclustering is needed. The reclastered part
includes pages 1, 2 and 3. Page 4 is not participating in the reclustering because its distance from page 1 is two steps which is bigger than the \textit{Radius} value of 1. Figure 3.4(b) shows the disk state right after the reclustering. Note that \textit{IIC} and \textit{EIC} in pages 1 and 2 are zero, because they have not yet been traversed as of the last reclustering.
(a) Before reclustering (triggering part).

(b) After reclustering.

Figure 3.4: Reclustering Algorithm.
Chapter 4

Experimental Evaluation

(Simulated Disk)

This chapter presents the experimental work that was performed in order to evaluate the PIXSAR system. We compared PIXSAR to DFS, a depth-first scan and store scheme. DFS is a natural clustering algorithm that scans an XML document in a depth-first search manner and assigns every encountered node to the currently used disk page. As soon as the current page is full, DFS starts assigning nodes to a new page. A major advantage of DFS is that it places together XML nodes that are neighbors in the document. Another advantage of DFS is that it is an online single-pass algorithm. DFS uses only parent-child edges, performs only a single initial clustering of the document, and then does not change the storage arrangement.

For experimental purposes, we used the XMark benchmark software [23] to
produce XML documents of different sizes. We chose representative XPath
queries (based on queries proposed in the XMark project). All experiments
were run on a x86-based Linux machine with 3GB main memory and 250 GB
disk space. Our implementations of PIXSAR and DFS employ an incremental
reference-counting garbage collector to aggressively detect and collect "dead"
objects. We used the number of page faults and the final partition cut value
(total of inter-page edge weights) as the main metrics of performance. The
space overhead for enabling reclustering was not very significant.

4.1 Main Parameters

To evaluate the performance of PIXSAR, we experimented with the following
parameters:

- Document size (DS) - 10MB, 124MB, 256MB.
- Cache size (CS) - approximately 5% and 10% of file size on disk.
- Reclustering radius (RR) - 2 and 3. A radius that is bigger than 3
  almost always caused total reclustering.
- Reclustering page limit (RPL) - a limit on the number of pages that
can participate in a reclustering. RPL is chosen to be 1% and 1.5% of
document size in pages.
- Reclustering factor (RF) - The magnitude of change to edge weights
  that triggers reclustering. We used three different factors: 1%, 2% and
  3% of change in the value of edge weights counters in the page (see
section 3.3).

4.2 Experiments Description

4.2.1 Setting Up

An experiment run has two input files: an XML document, and a log file with traces of queries to be run on the given document. The structure of this log file is as follows. There is a core composed of different queries which are called the basic log workload. This basic log workload is repeated a number of times. This creates the experimental workload. By using a basic workload that is repeated, the structure of the log roughly models a real-life workload. An experiment begins with loading the input document to the simulated disk. We use the DFS algorithm for the initial data placement on disk. Initially, every edge in the document has weight 1.

All logs were recorded by running the following basic queries. PARAM represents a randomly chosen value (chosen individually per occurrence). The original queries, and the relevant DTD, may be found at www.xml-benchmark.org.

- /site/people[position()=PARAM]/person[position()=PARAM]/name/text()
- /site/open_auctions[position()=PARAM]/open_auction[position()=PARAM]/bidder[position()=PARAM]/increase/text()
- /site/open_auctions[position()=PARAM]/open_auction[position()=PARAM]/bidder
4.2.2 Experiment Description

Every experiment has two main runs:

- **Incremental Run** - After loading the document to the simulated disk, we traverse the document according to the queries log file. During this traversal, we run the incremental algorithm as described in section 3, and collect the relevant statistical information (number of page faults and partition values) to result logs. Page faults occurring while reclustering are also counted.

- **DFS Run** - After loading the document to disk, it is traversed according to the queries log file. No changes are performed to the document placement on disk during this step. During the traversal, the relevant information is collected to result logs.
In order to easily comprehend the experimental results, we collect the relevant information 300 times during each run. We divide the traversal log into 300 equal parts, and at the end of each such part we log the relevant execution information. For example, if the log length is 30,000 then before starting traversing each sequence of 100 queries, we log information at this point.

4.3 Evaluation of PIXSAR

Our main goal is to examine how PIXSAR affects the number of disk page operations. Page operations (read or write) can occur in the following cases: Read-page while traversing the document (both in DFS-runs and in Incremental-runs); Write-page due to updating of edge weights while traversing the document (only in Incremental-Runs); Read/Write-page, during reclustering operations (only in Incremental-Runs).

PIXSAR is evaluated by counting only Read-Page operations. The reasons are as follows:

- Practically we can nearly eliminate write operations. The idea is that edge weights are updated probabilistically, e.g., every \( n \)’th update on the average for a parameter \( n \), rather than upon each and every change. This means that edge weights will vary more slowly, while still reflecting the workload, and that the number of write-page operations will be reduced significantly.
- Reclustering operations are preferably performed during "quiet peri-
ods”, i.e., when the system is lightly loaded with user operations.

- In case no such quite periods exist, the write operations incurred during reclustering can be budgeted (amortized) based on accumulated ”savings” in read operations.

In fact, for experimental convenience, we count all read-page-faults that occur during each run, including page faults that occur during the initial organization of the file, traversal and reclustering operations.

We experiment with the 10MB document with a log file size of 40,000 queries, while the basic log workload consists of 1000 queries (section 4.2.1). The results are presented in Table 4.1. We observe that for RPL=55, RR=2, and RF=2, PIXSAR reduces the number of page faults by about 50% as compared to DFS. We also note that: cache size (in the checked range) has little influence on the results; a larger RF provides better results; RPL is a dominant parameter. RPL=55 gives impressive results for RR=2 but very poor results for RR=3. The reason is that RPL=55 is too small for RR=3, because the reclustered portion is not inclusive enough, leading to very poor results. Recall that RPL is either 1% or 1.5% of document size in pages.

We ran experiments with a 124MB document with a log file size of 40,000 queries with basic log workload size of 1000 queries. In these experiments, we ran PIXSAR only over the first 30,000 queries. For the last 10,000 queries we performed no reclustering. The last stretch of 10,000 is simply run in order to ascertain that indeed the final layout of incremental run is better than that of DFS. In these experiments we used the knowledge gained during the 10MB
experiments. So, we excluded RR=3. We experimented with higher RF values to check if our thesis that larger RF values yield better results. In light of our previous observations, in these experiments we set RR=2 and varied only three parameters: RPL, RF and cache size. The results are presented in Table 4.2. We note that for RPL=400 and RF=3, PIXSAR provides 29% improvement. This is especially interesting as we performed reclustering operations only during the first 30,000 queries out of the 40,000 queries. These experiments show that higher RF values provide better results, and that cache size (at the range we checked) does not significantly affect the result.

![Graph comparing page faults for document size 10MB](image.png)

Figure 4.1: Comparing page faults for document with size 10MB. The x axis is the number of queries executed in units of 40,000/300.

Figures 4.1 and 4.2 show the behavior of PIXSAR in comparison to DFS in
terms of number of page faults. Figure 4.1 shows the result of an experiment with RR=2, RF=2, RPL=55 and cache size 5% of hard disk size. We observe that starting from \( x = 170 \), the slope of the PIXSAR curve stabilizes. This means that the number of page faults per unit time is fixed for the basic workload and is no longer being reduced by PIXSAR. Intuitively, PIXSAR "learned" the new workload. Figure 4.2 shows the result of an experiment with RR=2, RF=3, RPL=400 and cache size 5%. We observe that starting from \( x = 225 \), the slope of the PIXSAR curve stabilizes. But \( x = 225 \) is exactly 3/4 of the queries log size, i.e., the point after which no reclustering operation is performed. So, it is almost certain that had we continued to perform reclustering operations, the slope of the PIXSAR curve would continue...
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Page Faults</th>
<th>Cut Value</th>
<th>Impr(%)</th>
<th>Incremental</th>
<th>Impr(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR=2, RPL=40, CS=5%, RF=0.5</td>
<td>7,964,736</td>
<td>8,801,173</td>
<td>31</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>RR=2, RPL=40, CS=5%, RF=2</td>
<td>6,968,773</td>
<td>7,766,214</td>
<td>39</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>RR=2, RPL=40, CS=10%, RF=0.5</td>
<td>7,943,671</td>
<td>88,01,173</td>
<td>31</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>RR=2, RPL=40, CS=10%, RF=2</td>
<td>6,948,453</td>
<td>7,766,214</td>
<td>39</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>RR=2, RPL=55, CS=5%, RF=0.5</td>
<td>6,662,525</td>
<td>7,462,371</td>
<td>42</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>RR=2, RPL=55, CS=5%, RF=2</td>
<td>5,631,677</td>
<td>6,079,037</td>
<td>51</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>RR=2, RPL=55, CS=10%, RF=0.5</td>
<td>6,638,742</td>
<td>7,462,371</td>
<td>42</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>RR=2, RPL=55, CS=10%, RF=2</td>
<td>5,616,958</td>
<td>6,079,037</td>
<td>51</td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>RR=3, RPL=55, CS=10%, RF=0.5</td>
<td>9,386,466</td>
<td>10,418,588</td>
<td>18</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>RR=3, RPL=55, CS=10%, RF=2</td>
<td>9,271,841</td>
<td>10,614,429</td>
<td>19</td>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>

Table 4.1: Results for a 10MB document. In all the experiments, the number of DFS page faults is 11,466,303, the DFS cut value is 13,483,580. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run.

to evolve, which means that the disk placement quality of the document can perhaps still be improved.

We ran experiments over a 256MB document with a log file size of 40,000 queries. The basic log workload size is about 700 queries, this number was chosen because of the enormous sizes of workload log files (generated by
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Page Faults</th>
<th></th>
<th>Cut Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incremental</td>
<td>Impr(%)</td>
<td>Incremental</td>
<td>Impr(%)</td>
</tr>
<tr>
<td>RR=2, RPL=400, CS=5%, RF=2</td>
<td>97,369,219</td>
<td>26</td>
<td>1.01E+08</td>
<td>33</td>
</tr>
<tr>
<td>RR=2, RPL=400, CS=5%, RF=3</td>
<td>90,967,337</td>
<td>29</td>
<td>92,994,035</td>
<td>35</td>
</tr>
<tr>
<td>RR=2, RPL=400, CS=10%, RF=2</td>
<td>97,334,759</td>
<td>24</td>
<td>1.01E+08</td>
<td>29</td>
</tr>
<tr>
<td>RR=2, RPL=400, CS=10%, RF=3</td>
<td>90,942,293</td>
<td>29</td>
<td>92,994,035</td>
<td>35</td>
</tr>
<tr>
<td>RR=2, RPL=600, CS=5%, RF=2</td>
<td>96,054,414</td>
<td>25</td>
<td>99,986,869</td>
<td>30</td>
</tr>
<tr>
<td>RR=2, RPL=600, CS=5%, RF=3</td>
<td>94,043,216</td>
<td>26</td>
<td>96,153,682</td>
<td>33</td>
</tr>
<tr>
<td>RR=2, RPL=600, CS=10%, RF=2</td>
<td>96,012,825</td>
<td>25</td>
<td>99,986,869</td>
<td>30</td>
</tr>
<tr>
<td>RR=2, RPL=600, CS=10%, RF=3</td>
<td>94,000,952</td>
<td>26</td>
<td>96,153,682</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 4.2: Results for a 124MB document. In all the experiments, the number of DFS page faults is 1.28E+08, the DFS cut value is 1.42E+08. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run.
Saxon runs). In these experiments as well, we ran PIXSAR only on the first 30,000 queries. For the last 10,000 queries no reclustering was performed (the motivation for that is the same as in experiments with 124MB document). In these experiments, using the intuition gained in previous experiments, we excluded RR=3 and cache size 5%. We also experimented with RF=4 in order to check whether a large value for RF is beneficial for large files. The results are presented in Table 4.3. We note that for RPL=1500 and RF=4, PIXSAR provides a 14% improvement. The result is less impressive, probably because the basic workload size and the workload file itself are not large enough. Their sizes were chosen due to limited resources. All the experiments exhibit a high correlation between a low cut value and a low number of page faults. In the 256MB experiments, we observe that page fault improvement is about 14% and the cut value improvement is 16% whereas in the 10MB experiments, the page faults improvement and the cut value improvement are 35%-50%.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Page Faults Incremental</th>
<th>Impr(%)</th>
<th>Cut Value Incremental</th>
<th>Impr(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR=2, CS=10%</td>
<td>2.14E+08</td>
<td>13</td>
<td>2.26E+08</td>
<td>16</td>
</tr>
<tr>
<td>RPL=1000, RF=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RR=2, CS=10%</td>
<td>2.12E+08</td>
<td>14</td>
<td>2.28E+08</td>
<td>15</td>
</tr>
<tr>
<td>RPL=1500, RF=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Results for a 256MB document. In all the experiments, the number of DFS faults is 2.45E+08, the DFS cut value is 2.77E+08. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run.
Indexes: We experimented, with a 10 MB document, with queries that utilize indexes. Each experiment was run over a different log. Each such log contains repetitions of one of the following indexed queries for 5000 times. Other parameters were fixed to RPL=55, RR=2 and RF=3. The queries used are:

1. `/site/open_auctions/open_auction[position() = 96]/bidder[1]/increase/text()`
2. `/site/open_auctions/open_auction'/initial/text()`
3. `/site/closed_auctions/closed_auction[position() = 235]/price/text()`
4. `/site/regions/*/item[position() = 60]/*/text()`

The ‘’’ indicates an index usage. The results are presented in Table 4.4. We observe that in three queries out of four there is an improvement of about 40% in the number of page faults, which is similar to the improvement in queries without using indexes while running with the same values for RPL, RR and RF. For the second query, the improvement is only 2%. This query nearly navigates the whole document. So, smart packing has little impact on performance and the query endures additional page faults during reclustering operations. We conclude that for queries that traverse a large portion of the data, PIXSAR improves little or not at all.

In another experiment, we ran PIXSAR on a workload log file and then, after processing 20,000 queries, we changed to a different workload log file. The goal of this experiment was to examine how PIXSAR adjusts to a completely new workload after having fitted storage placement for an initial workload.
<table>
<thead>
<tr>
<th>log</th>
<th>Page Faults</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incremental</td>
<td>DFS</td>
<td>Impr(%)</td>
</tr>
<tr>
<td>log of 1</td>
<td>18,284</td>
<td>27,111</td>
<td>33</td>
</tr>
<tr>
<td>log of 2</td>
<td>4,306,586</td>
<td>4,394,236</td>
<td>2</td>
</tr>
<tr>
<td>log of 3</td>
<td>12,712</td>
<td>22,120</td>
<td>43</td>
</tr>
<tr>
<td>log of 4</td>
<td>43,583</td>
<td>77,021</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 4.4: Results for indexed queries. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run.

For this experiment we use two logs with different sizes. One log has 20,000 queries while the basic workload size is 500 queries. The second log has 40,000 queries with a basic workload size of 1000 queries. A first run executed PIXSAR for both logs (concatenated). A second run executed PIXSAR only for the first log and while processing the second log no reclustering operations were performed. The RPL, cache size and RR parameters are fixed and the RF parameter is varied. The results are presented in Table 4.5. The conclusion is that PIXSAR adjusts well to a drastic change in the workload.

It may be interesting to examine the case of a slowly changing workload.

The exact connection between observed improvements to file characteristics is not clear. In almost all cases a significant improvement is observed. In a set of experiments conducted over synthetically created files, with various "profiles", significant improvements, albeit smaller, were observed. For 10MB files the number of page faults is smaller by at least 20%; for 124MB files the number of page faults is smaller by about 10%.
<table>
<thead>
<tr>
<th>RF value</th>
<th>Page Faults</th>
<th></th>
<th></th>
<th>Impr(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First run</td>
<td>Second run</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RF=2</td>
<td>16,867,725</td>
<td>23,564,931</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>RF=3</td>
<td>15,935,473</td>
<td>23,706,331</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>RF=4</td>
<td>14,993,078</td>
<td>23,746,358</td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Results for the "changing workload" experiment. The Impr columns show percentage improvement of the first run in comparison to the second run.
Chapter 5

Experiments with a Real
(Physical) Disk

After obtaining promising results in experiments with simulated disk, we performed experiments with a physical (real) disk. We experimented with a disk with the following characteristics: capacity - 500 GB, rotational speed: 7,200 rpm, cache - 16 MB, interface - SATA II, max. external transfer rate - 300 MB/s, seek time read - 8.9 ms, seek time write: 10.9 ms, seek time track-to-track - 2.0 ms, seek time full stroke (the amount of time to seek the entire width of the disk, from the innermost track to the outermost): 21.0 ms.

The structure of the experiments is exactly as described in section 4, i.e., we compare the performance of the PIXSAR algorithm to that of DFS. To get as close as possible to modeling a real world scenario, we have used the

56
following model. There are several users that concurrently use the disk, and
some of them perform disk write operations. Specifically, every $t$ units of
time, on the average, a write operation is performed ($t$ is parameter).

## 5.1 Main Parameters

We compare the performance of DFS and PIXSAR by counting only the time
of traversals over the document caused by query tasks (but not the time of
reclustering operations - which are regarded as server’s operations). We also
compare the total run time of both Algorithms. Time is measured in seconds.
Each experiment is characterized by settings to parameters. The following
parameters were set to fixed values (based on experimentation). Cache size
was set to 10% of disk size. Reclustering page limit (a limit on the number
of pages that can participate in a reclustering) was set to 1.5% of document
size in pages. Other parameters were set as follows:

- Document size (DS) - 10MB, 20MB, 30MB.
- Reclustering radius (RR) - 2 and 3.
- Reclustering factor (RF) - The magnitude of change to edge weights
  that triggers reclustering. We used two different factors: 2% and 3%
  of change in the value of edge weights counters in the page (see sec-
  tion 3.3).
• Disturbance time (DT) - Percentage of write operations performed to a random location in the disk. For example, DT=0.5, means that 0.5% of the read operations, on average, are followed by a write operation to a randomly chosen location in the disk. We used the following DT values: 0%, 0.5%, 0.66% and 1%.

• Edge Update Frequency (EUF) - Percentage of updating edge weights. Edge weights are updated probabilistically, e.g., every n’th update on the average for a parameter n, rather than upon each and every change. For example EUF=20 means that we update an edge weight only in 20% of edge traversal cases.

• Number of queries in the queries log file (QN) - 150,000, 200,000, 400,000 and 600,000.

Two parameters deserve a detailed explanation. EUF represents a tradeoff between accuracy and practicality. Each time an edge weight need be incremented, the operation is actually performed on disk and consequently the system will be dominated by a large number of disk write operations and the result will be performance degradation rather than improvement. So, rather than performing each incrementation, the decision as to whether to perform the operation on disk is obtained probabilistically. This has the advantage of not having to ”remember” the number of such incrementations, to eventually represent the workload faithfully, and yet perform few actual write operations for updating edge weights. The DT parameter has a very
different role. It models "other activities" on the physical disk by applications. These other activities naturally cause the disk head to move to various cylinders of the disk. Not taking such operations into account is valid only for a disk that is under the control of a single thread of a single application. Since we assume that the disk may support other threads of applications, the need to model their effect on disk head location is crucial. If this is not modeled, DFS would appear to be highly superior, which is the case only for highly restricted environments.

5.2 Results of Experiments

Our first set of experiments had no writing disturbances at all (DT is zero). We tried to vary other parameters, but the results were the same. DFS’ performance is approximately 6 times better than that of PIXSAR. The conclusion is that when there are no write operations to the disk, DFS is superior. The reason is that a typical disk has a small cache (our disk cache size is 16MB). The document size in the experiments was 10MB, which is less than the disk cache size. Due to the fact that DFS does not perform any disk write operations, the disk, except for its own cache, is hardly used. Simply, the whole document is loaded into the cache, and all the traversal operations are performed in cache. In contrast, PIXSAR, while traversing the document, updates edge weights, i.e., continuously performs write operations to the disk. These operations are performed straight to the disk and not to
Table 5.1: Results for a 10MB document. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. In all the experiments, the number of DFS page faults is 57,400,167.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Page Faults</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incremental</td>
<td>Impr(%)</td>
</tr>
<tr>
<td>DT=1</td>
<td>23,555,706</td>
<td>59</td>
</tr>
<tr>
<td>DT=0.66</td>
<td>23,389,912</td>
<td>59</td>
</tr>
<tr>
<td>DT=0.5</td>
<td>23,160,833</td>
<td>60</td>
</tr>
</tbody>
</table>

the cache (to avoid crash recovery problems). So, the disk is used directly, which is time consuming.

In the second set of experiments, only the DT parameter is varied and the other parameters are set to values as follows: RR=2, RF=3, EUF=20 and QN=150,000. The results are presented in Table 5.1. PIXSAR greatly outperforms DFS with more than 60% improvement. We observe that there are almost no differences in the three results, meaning that the DT parameter has to be varied much more in order to see some difference.

In the third set of experiments, DT was set at 0.66%, and the other parameters were varied. The results are presented in Table 5.2. The improvement in number of page faults is consistent with the improvement in actual elapsed time. In Figure 5.1 we show the behavior of PIXSAR in comparison to that of DFS in terms of number of page faults and of elapsed time. It shows the results of an experiment with RR=2 & 3 (randomly, in half of the reclustering events it was 2 and in the other half, 3), RF=3, DT=0.66%, EUF = 10% and QN=600,000. We observe that, in both graphs, starting from \( x = 180 \), the
Table 5.2: Results for a 10MB document. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Page Faults</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incr</td>
<td>DFS</td>
</tr>
<tr>
<td>RR=2.3, EUF=20%, QN=200,000</td>
<td>42,089,104</td>
<td>57,400,167</td>
</tr>
<tr>
<td>RR=2.3, EUF=10%, QN=600,000</td>
<td>62,064,475</td>
<td>172,196,787</td>
</tr>
</tbody>
</table>

slope of the PIXSAR curve stabilizes. This means that the number of page faults per unit time is fixed for the basic workload and is no longer being reduced by PIXSAR. Intuitively, PIXSAR has "learned" the workload.

We conducted more tests with different document sizes. The results are presented in Table 5.3. In these experiments we also see a significant improvement of PIXSAR over DFS.

An interesting observation concerns the total run time improvement (including all operations for both querying and reclustering) of PIXSAR in comparison to DFS, under the disturbance model that we use. Note that in the first line, we see improvement of 0%. The reason is that EUF was 20% and NQ was only 200,000. So many edge weight updates were performed, causing a lot of time costly reclustering operations, while the number of queries is relatively small. In order to see the total time improvement, the test had to run much longer, as shown in the other tests. In all other tests, PIXSAR run time outperforms DFS by at least 30%! The results are presented in Table 61.
Figure 5.1: Comparing DFS with PIXSAR for document with size 10MB. The $x$ axis is the number of queries executed in units of 600,000/1000.
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Page Faults</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incr</td>
<td>DFS</td>
</tr>
<tr>
<td>DS=20MB, QN=354,000</td>
<td>84,837,190</td>
<td>196,807,553*</td>
</tr>
<tr>
<td>DS=30MB, QN=400,000</td>
<td>148916146</td>
<td>328078427*</td>
</tr>
</tbody>
</table>

Table 5.3: Results for documents with other sizes. The Impr columns show percentage improvement of an incremental run in comparison to a DFS run. The DFS run in the first experiment was performed for 100,000 queries only and for 160,000 in second experiment, so the numbers in the table that are marked with "*" were extrapolated which is justified as the DFS line is always straight.

5.4.

The results of experiments with a real physical disk, reported herein, are surprisingly better than those with a simulated disk. The main reason for this is the small number (caused by the probabilistic scheme) of edge weights updates. As a result of that, less reclustering operations occur, but each reclustering operation is much more significant. So, the reorganization of the disk is better, and this causes a smaller number of read page faults as compared to DFS. Another important factor is the disturbance model.
<table>
<thead>
<tr>
<th>Test Parameters</th>
<th>Time Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS=10MB, EUF = 20%</td>
<td>0</td>
</tr>
<tr>
<td>QN=200,000</td>
<td></td>
</tr>
<tr>
<td>DS=10MB, EUF = 10%</td>
<td>57</td>
</tr>
<tr>
<td>QN=600,000</td>
<td></td>
</tr>
<tr>
<td>DS=20MB, EUF = 10%</td>
<td>39</td>
</tr>
<tr>
<td>QN=354,000</td>
<td></td>
</tr>
<tr>
<td>DS=30MB, EUF = 10%</td>
<td>32</td>
</tr>
<tr>
<td>QN=400,000</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Time improvement, showing percentage improvement of an incremental run in comparison to a DFS run.
Chapter 6

iPIXSAR

This chapter describes the extensions to PIXSAR’s algorithms in order to take indices into account during incremental partitioning.

First we define some of the terminology we use:

- **Index**: is a set of index entries. The collection is based on an XPath index expression. For example, an index $X$ may be based on the XPath expression $/a/b/c$.

- **Index entry**: is an entry in the index set with a pointer to a node that conforms to the index expression. Continuing the example, an index entry $e$ in index $X$ would include a pointer to a $c$-labeled node, say $v$, in the XML document, whose parent is a $b$-labeled node that has an $a$-labeled parent that is a child of the root.

- **Index Target Node**: A node that is pointed by an index entry of a particular index is called a target (XML) node of that index entry.
Node \( v \) in the example is an index target node.

iPIXSAR incrementally improves the placement of the document on disk based on the actual querying activity (workload). iPIXSAR addresses a new problem that, as far as we know, was never investigated, the clustering of multi rooted trees (MRTs). Indices are modeled by a MRT in which each root corresponds to a distinct index and index target nodes define the subtrees of the document tree in the tree to which this index apply. iPIXSAR therefore addresses the effects of indexing (not addressed at all by PIXSAR), and takes into account the affinity between nodes that are target nodes of subsequent index entries of some index, these \textit{do not} necessarily have a (structural) parent-child or sibling edge between them.

All edges in the document graph are unidirectional. The direction is from a node to its parent, from a node to its left sibling and from an index target node to the immediately previous index target node.

For simplicity of presentation, assume we have a single index, say \( X \), which is based on a XPath expression \( exp \). Let \( e_1, \ldots, e_n \) be the index entries and let \( x_1, \ldots, x_n \) be the corresponding index target nodes. Each \( e_i \) also contains the weight of the virtual edge \((x_i, x_{i+1})\), \( 0 < i \leq n \). iPIXSAR extends PIXSAR and relies on the same infrastructure (see 2).

The intuition underlying iPIXSAR is as follows:

- Select trees out of the full document graph. The trees include the one corresponding the basic XML hierarchical structure as well as trees implied by indices.
• Determine the overall "profit" in cut value that PIXSAR can obtain based on each of these trees.

• Perform reclustering for only the most "profitable tree".

iPIXSAR extends PIXSAR in its way of updating the edge weights and in the reclustering operation itself (incrementalRecluster algorithm).

6.1 Index-based Incremental Reclustering

We use two parameters in this algorithm. The first is reclustering flag, which is set to on as soon as we determine that reclustering is needed (it is initially set to off). The second is reclustering node, which holds the node that caused the reclustering decision.

The IdxIncrementalR algorithm is presented in Figure 6.1. The function runIncrRecluster(v) in line 1.2 is described later on and appears in Figure 6.3. The motivation of the "if" in line 5 is to reduce the number of times in which we check whether reclustering is needed. The function toRecluster(v) in line 5.0 is explained in section 3.3. The motivation of line 5.1 is to provide an indication of the necessity of a reclustering for use in line 1.0. To describe the runIncrRecluster(v) function we need the following definitions:

• Base augmented tree: The XML document tree augmented with sibling edges. That is, the root of this tree is the document root.
For each traversed node \( v \):

1. if (\( v \) is first scanned node of the query)
   1.0 if (\( rFlag = on \))
   1.1 \( rFlag = off \)
   1.2 \( runIncrRecluster(v) \)

2. Update the weight of all edges that connect \( v \) to other nodes that are "close enough" by using the TPA algorithm.

3. if (\( v \) is an index target node \( x_i \) of some index \( X \)) /*observe there may be several such indices. */
   3.0 update the weight of the \((x_i, x_{i-1})\) edge provided \( x_i \) and \( x_{i-1} \) are "close enough".

4. Update the incremental weight counters of \( v \)'s page.

5. if (\( random() < q \)).
   5.0 if (\( toRecluster(v) \))
   5.1 if \( rFlag = off \)
   5.2 \( rFlag = on \)
   5.3 \( rNode = v \)

Figure 6.1: The \texttt{IdxIncrementalR} Algorithm.
• **X index augmented tree**: X is an index. The root of the tree is a dummy node with the name of the corresponding index, X (note that the root node of this tree does not exist in the disk). The children of this X node are all the index target nodes of X. Every two sibling index target nodes \(x_i\) and \(x_{i-1}\) have an edge between them. The subtree rooted at each index target node \(x_i\) is exactly as in the base augmented tree.

• **Full graph**: The base augmented tree unioned with all index augmented trees.

• **Gain in cut value**: The difference in cut value, before and after a reclustering operation, in the full graph. Note that because root nodes of index augmented trees are dummy nodes, we do not count the edge weights of edges connected to these root nodes while calculating the gain in cut value.

Figure 6.2 shows all the possible trees for the document appearing in figure 6.2(a). There is one index named X that corresponds to the expression: ”/root/a/b”. Observe that in general there may be additional indices. Three variables are used in the `runIncrRecluster(v)` function. The first is `maxGain`, which holds the maximal found gain in cut value (it is initially set to 0). The second is `maxTreePointer`, which holds a pointer to a tree that provides the maximal gain in cut value. The third is `indicesList`, which contains the list of relevant indices.

The `runIncrRecluster(v)` function is presented in Figure 6.3. The `max(x, y)`
function in line 3.3.5 returns the maximum between $x$ and $y$.

### 6.2 Example

To illustrate the iPIXSAR algorithm, we present an example in Figure 6.2. For this example, we define the page size to be 3 (say KB), node size to have a fixed size of 1, $TRF$ to be 2% and $reclusteringRadius$ as 1. The left side of Figure 6.4(a) shows the disk state after running initial clustering, and before
1. Build $S$, a subtree of the base augmented tree that has to participate in the reclustering according to the given radius and other parameters. During building:
   1.0 Fill $indicesList$ with indices names corresponding to index target nodes which are part of $S$.

2. $maxTreePointer = pointerTo(S)$.

3. if ($indicesList$ is not empty)
   3.0 Convert $S$ into the procedure $XS$ internal data format, $IS$.
   3.1 Run $XS$ on $IS$.
   3.2 $maxGain =$ the gain in cut value that $XS$ achieves on $IS$.
   3.3 For each $X$ in $indicesList$:
      3.3.0 Build $I$, the part of the corresponding $X$ index augmented tree that has to participate in the reclustering.
      3.3.1 Convert $I$ into the procedure $XS$ internal data format, $II$.
      3.3.3 Run $XS$ on $II$.
      3.3.4 $tmpGain =$ the gain in cut value that $XS$ achieves on $II$.
      3.3.5 $maxGain = max(maxGain, tmpGain)$.
      3.3.6 if (equal(maxGain, tmpGain))
      3.3.7 $maxTreePointer = pointerTo(I)$.

4. Convert the tree $maxTree$ pointed by the $maxTreePointer$ into the procedure $XS$ internal data format, $IMT$.

5. Run $XS$ on $IMT$.

6. Assign each resulting cluster to a page having sufficient space to hold it. During each assignment do as follows:
   6.0 Write the current state of the page (internal and external) weights into the page counters.
   6.1 Add the current global counter to the global counters array.
   6.2 Write the ID of the current global counter to the page.

Figure 6.3: Function runIncrRecluster.
starting to traverse the document. We observe that the $IPC$ and $EPS$ counters contain the corresponding initial values, while $IIC$ and $EIS$ are zero, because the traversal has not yet started. The right side of Figure 6.4(a) shows the disk state right after traversing edge $(a, b)$ (the rightmost $b$), i.e., while visiting node $b$. $IIC$ and $EIC$ are updated and contain the difference between the current and the initial values of the page’s counters. In the next step, we run the triggering test on page 1 (the page in which node $b$ resides) and discover that the percent of $IPC$ out of $GC_{OLD}$ is 30% ($13/44$) while $IIC + IPC$ out of $GC_{NEW}$ is 27% ($23/85$). Since $27/30 = 0.90 < 0.98$, triggering occurs. So, reclustering is needed. The part that is reclustered includes pages 1,2 and 3. Page 4 does not participate in the reclustering because its distance from page 1 is two steps which is greater than the $Radius$ value of 1. The reclustering portion in the base augmented tree contains index target nodes. So, we check the gain in cut value for two trees (the base augmented tree and the $X$ index augmented tree). Figure 6.4(b) shows the optional reclustering based on the base augmented tree. Figure 6.4(c) shows the optional reclustering based on the $X$ index augmented tree. We find that the gain in cut value for the base augmented tree is 3 ($40 - 37$) while for the $X$ index augmented tree it is 12 ($40 - 28$). So, we perform reclustering on the $X$ index augmented tree (Figure 6.4(c)). Note that $IIC$ and $EIC$ in pages 1,2 and 3 (the pages that participate in the reclustering operation) right after the reclustering are still zero, since they have not yet been traversed as of the last reclustering.
(a) Before reclustering (triggering part).

(b) After reclustering using the base

(c) After reclustering using the X index

augmented tree.

Figure 6.4: Index Reclustering Algorithm.
6.3 Evaluation of iPIXSAR

For experimental purposes, in order to create an interesting indices based examples, we wrote a simple XML document generator (the DTD of the generator is shown in Appendix B). We used this generator to create test cases with indices. We used the number of page faults and the final partition cut value (total of inter-page edge weights) as the main metrics of performance. These experiments were executed on a simulated disk. We count all read-page-faults that occur during each run, including page faults that occur during the initial organization of the document, traversal and reclustering operations.

To evaluate the performance of iPIXSAR, we experimented with document sizes of 10MB and 124MB.

The following parameters were set to fixed values (based on extensive experimentation):

- Cache size was set to 10% of document size in disk.
- Reclustering page limit (a limit on the number of pages that can participate in a reclustering) was set to 1.5% of document size in pages.

Other parameters were set as follows:

- Reclustering radius (RR) - 2 and 3.
- Reclustering factor (RF) - The magnitude of change to edge weights that triggers reclustering. We used two different factors: 2% and 3% of change in the value of edge weight counters in the page (see section 3.3).
Number of different indices (NOI) - 1 and 2.

We used the following index expression:

- "collection/cake_recipe/ingredient_complex/ingredient"
- "collection/bread_recipe/ingredient_complex/ingredient"
- "collection/meat_recipe/ingredient_complex/ingredient"

All logs were recorded by running the following basic queries. PARAM represents a randomly chosen value (chosen individually per query occurrence), the index part is in bold.

- /collection/cake_recipe/ingredient_complex/ingredient\text{/preparation} /step\_PARAM/text()
- /collection/cake_recipe/*[position()=PARAM]/preparation/step\_4/text()
- /collection/bread_recipe/ingredient_complex/ingredient\text{/preparation} /step\_PARAM/text()
- /collection/*[position()=PARAM]/nutrition/@fat
- /collection/*/*[@unit='cup']/ingredient_complex/*[position()=PARAM]/preparation /step\_PARAM/text()
- /collection/meat_recipe/ingredient_complex/ingredient\text{/preparation} /step\_PARAM/text()
- /collection/meat_recipe/ingredient_complex/ingredient_complex/ingredient_complex /preparation/step\_1/text()
The experiments were done in a manner similar to the PIXSAR experiments (see section 4).

In the first set of experiments, we used a 10MB document. The queries log file was 160,000 queries long. The results are presented in Table 6.1. PF stands for page faults. The improvement of iPIXSAR over PIXSAR varies between 3% to 8%. For Example, in experiment with RR=3, RF=2 and NOI=3, iPIXSAR provides an improvement of 8.2% over PIXSAR. Another interesting result is the experiment with RR=3, RF=2 and NOI=3 (i.e., with 3 different indices!), where the improvement of iPIXSAR over PIXSAR is 6.1%. In Figure 6.5 we show the behavior of iPIXSAR in comparison to that of PIXSAR in terms of number of page faults. It shows the results of the experiment with RR=3, RF=2 and NOI=1. We observe that, the slope of iPIXSAR grows slower than that of PIXSAR. Also, in both graphs, starting from $x = 180$, the slopes of both curves stabilize. This means that the number of page faults per unit time is fixed for the basic workload and is no longer being reduced by both algorithms. Intuitively, they have "learned"
### Table 6.1: Results for a 10MB document. PF columns show number of page faults. The Impr1 column show percentage improvement of a PIXSAR run in comparison to a DFS run. The Impr2 column show percentage improvement of an iPIXSAR run in comparison to a PIXSAR run.

<table>
<thead>
<tr>
<th>Experiment Parameters</th>
<th>DFS PF</th>
<th>PIXSAR PF</th>
<th>iPIXSAR PF</th>
<th>Impr 1 (%)</th>
<th>Impr 2 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOI=1, RR=2, RF=2</td>
<td>34,911,033</td>
<td>28,094,417</td>
<td>26,169,319</td>
<td>20</td>
<td>6.9</td>
</tr>
<tr>
<td>NOI=1, RR=2, RF=3</td>
<td>34,911,033</td>
<td>28,393,899</td>
<td>26,270,965</td>
<td>19</td>
<td>7.4</td>
</tr>
<tr>
<td>NOI=1, RR=3, RF=2</td>
<td>34,911,033</td>
<td>28,247,853</td>
<td>25,933,148</td>
<td>19</td>
<td>8.2</td>
</tr>
<tr>
<td>NOI=1, RR=3, RF=3</td>
<td>34,911,033</td>
<td>28,286,191</td>
<td>26,282,519</td>
<td>19</td>
<td>7.1</td>
</tr>
<tr>
<td>NOI=2, RR=2, RF=2</td>
<td>39,385,785</td>
<td>33,443,175</td>
<td>32,338,100</td>
<td>15</td>
<td>3.3</td>
</tr>
<tr>
<td>NOI=2, RR=2, RF=3</td>
<td>39,385,785</td>
<td>33,308,135</td>
<td>32,789,606</td>
<td>15</td>
<td>1.6</td>
</tr>
<tr>
<td>NOI=2, RR=3, RF=2</td>
<td>39,385,785</td>
<td>32,703,068</td>
<td>31,445,995</td>
<td>17</td>
<td>3.8</td>
</tr>
<tr>
<td>NOI=2, RR=3, RF=3</td>
<td>39,385,785</td>
<td>32,714,216</td>
<td>31,584,575</td>
<td>17</td>
<td>3.5</td>
</tr>
<tr>
<td>NOI=3, RR=2, RF=2</td>
<td>43203016</td>
<td>35,154,715</td>
<td>34,302,104</td>
<td>19</td>
<td>2.4</td>
</tr>
<tr>
<td>NOI=3, RR=2, RF=3</td>
<td>43203016</td>
<td>35,866,092</td>
<td>34,865,094</td>
<td>17</td>
<td>2.7</td>
</tr>
<tr>
<td>NOI=3, RR=3, RF=2</td>
<td>43203016</td>
<td>35,236,096</td>
<td>33,066,640</td>
<td>18</td>
<td>6.1</td>
</tr>
<tr>
<td>NOI=3, RR=3, RF=3</td>
<td>43203016</td>
<td>33,908,089</td>
<td>33,290,717</td>
<td>22</td>
<td>1.8</td>
</tr>
</tbody>
</table>

the workload.

![Diagram](image)

Figure 6.5: Comparing page faults for a document of size 10MB. The x axis is the number of queries executed in units of 40,000/300. The y axis is the improvement of iPIXSAR in comparison to PIXSAR.
In the second set of experiments, we used a 124MB document. The results are presented in Table 6.2. The improvement of iPIXSAR over PIXSAR is less than 1%.

**Discussion:** The 10MB experiments with query file length of 160,000, with a single index show roughly an 8% improvement, which is significant. With two indices, the improvement is about 4%. With three indices, the improvement is about 2.5% (note that in one of the experiments with three indices the improvement is 6.1%). It is also possible that with multi-indices it takes longer to see the reclustering effects (intuitively, it takes longer to ”learn” a more complex behavior). Also, the more indices are taken into account, the more work needs to be done in deciding which index-induced tree should be reclustered by XS. In particular, more page faults may occur. So, with more indices present, we expect more overhead and possibly less savings in case there is no significant difference in the savings realized by the various index-induced augmented subtrees. The 124MB experiments do not reveal such a large improvement. In case of large files, more overhead is incurred simply because the number of pages that are examined by XS in whatever subtree it handles is proportional to the overall size of the file in pages. We have ideas on how to resolve both of these problems, and plan to experiment with them in the future. One idea is not to evaluate all the, say \(N\), subtrees of the base augmented tree and those implied by indices but rather to randomly choose a subset of size \(0 < k < N\) of them where \(k\) is a parameter. Similarly, for large files, perhaps the percentage of total size in pages should be capped.
Table 6.2: Results for a 124MB document. The Impr columns show percentage improvement of an iPIXSAR run in comparison to a PIXSAR run.

with a constant number of pages, the rationale is that changes are usually local irrespective of file size. So, indices do influence the effectiveness of XML storage. Since the same machinery is used for storage rearrangement with and without indices, it is beneficial to recluster while taking indices into account.
Chapter 7

System Architectures

7.1 Using PIXSAR in a single disk context

A system that uses XML data can utilize the PIXSAR algorithm, provided it has the following three (we shall refer to these as 1, 2 and 3) functionalities:

1. Updating edge weights - while traversing, from time to time, updating the workload information of the requested node and of its page (according to the TPA algorithm, as described in section 3.1).

2. Triggering reclustering operations - checking, once in a while, if the page of the currently requested node requires a reclustering operation (according to the incremental triggering algorithm as described in [27]).

3. Performing reclustering, when the need for reclustering is detected.

We offer a few basic designs which are presented in Figure 7.1, for integrating PIXSAR within a computing system.
Figure 7.1: Architecture schemes. (a) Self-Reclustering Disk (b) Clustering Operating System (c) Clustering Database.
Self-Reclustering Disk

This is a self-reclustering disk. We define two special units (see figure 7.1(a)). One is a part of the operating system, and is in charge of functionality 1. The other is located in the physical disk. It is in charge of functionalities 2 and 3. Functionality 2 has to be activated during write page-faults. That is, in some percentage of page write operations, the triggering algorithm is activated to check if reclustering is needed, then it performs reclustering (functionality 3) in case it is necessary. Functionality 1 cannot be located in the physical disk, as the disk does not "know" the way this data is used.

Clustering Operating System (OS)

This system is independent of the type of the physical disk it uses. All functionalities (1-3) are placed in one unit. This unit has to be integrated into the operating system (see figure 7.1(b)). This unit must have the right to "bypass" all cache levels and to issue direct commands to the physical disk in order to be able to reorganize it.

Clustering Database

Many database systems have to cope with handling large XML documents or other tree structured files. We consider a system that can organize its data on disk for optimizing performance. In this design, a stand-alone unit provides all three functionalities. This unit has to communicate with the operating system in order to realize functionality 1, and has to communicate with the disk in order to perform functionalities 2 and 3. As in the "clustering operating system" design, this unit must also be able to "bypass" all cache
levels in order to issue direct commands to the disk.

7.2 Extensions for Novel Architectures

There are many ways to optimize and extend the PIXSAR algorithm. We describe two such extensions.

7.2.1 Two Levels Mass Memory

Till now we considered a model which has only one level of mass storage memory. Here we consider systems that have two mass memory levels. In this model, the first mass memory level is fast but of limited capacity. The second level has practically unlimited capacity, but the streaming to and from the second level can be expensive in terms of both time and money.

There are two main concerns about two memory levels. The first concern is making sure that the most relevant data is placed in the first memory level, which is faster, whereas all other data (assumed less relevant) is held in the second memory level. The second concern is organizing the information in both levels so as to save on access time (and cost).

We view all the stored information as one ”big tree”, where some subtrees are located in the first memory level and the other parts of the tree are in the second memory level (see Figure 7.2).

We present systems models that capture the essence of two mass memory levels. In these models, the first memory level is a disk (or collection of
disks), whereas the second memory level is a remote storage facility. The models are characterized along three axes:

1. Storage mode on the second memory level. We differentiate between:

   - **BOM (Block Object Model)**: The size of each entity that is stored in the second memory level is one page.
   - **FOM (File Object Model)**: Each object that is stored in the second memory level is a file (of arbitrary size). Each file may hold many pages. On the one hand, this may reduce the number of requests that are needed in order to read large amounts of data. On the other hand, in case that we need only one page, we must read the whole file, which can be a costly operation.

2. Movement of data to/from the first level from/to the second level. In the models we consider this capability is essential and is motivated by a desire to store the most frequently used data in the first level.
3. Movement of data within pages of a certain level. In the models we consider, this capability is always available in the first level. This capability may or may not be available in the second level.

The third axis, movement of data within pages of a level, may be exceedingly expensive for slow devices. It may also be very expensive in certain settings. For example, consider the Amazon S3 system [26]. Amazon S3 (Simple Storage Service) is an online storage web service, that provides practically unlimited storage. The usage of S3 is not free of charge. Transferring 1GB into S3 costs $0.1, and out of S3 $0.17. Storage of 1GB costs $0.15 per month. Each S3 request is also charged, the amount depends on the type of the request. One more characteristic of S3 is that the user can not directly change a stored data unit. The only way to change a stored data unit is to read it completely to main memory and to update it there, then to store the updated data unit once again to Amazon S3. So, one can conceivably perform movement of data within pages, by reading them into main memory, or the first level memory, performing the reorganization there and then storing back to the second memory level. More information about S3 is available in [26].

We consider a settings (similar to Amazon S3) with BOM (axis 1), movement of data to/from the first level from/to the second (axis 2) is enabled, and movement of data among pages of the second level is not allowed (axis 3).

Next, we describe the Two Memory Levels algorithm. The goal of this algorithm is to decide which data is the most relevant at each moment (accessed frequently), and to perform all the needed operations that are required to
keep such data in the first memory level.

We define:

- The cut is the set of edges such that for each edge, one node is placed in the first memory level whereas the second node is placed in the second memory level (see Figure 7.3).

- A page is said to be above (respectively, below) the cut, if it resides in the first level memory but contains an element that is connected via an edge to an element in a page in the second (respectively, first) level memory.

- The value of the cut is said to be the sum of the weights of all the edges in the cut.

The Two Memory Levels algorithm treats pages. The basic idea underlying this algorithm is to check, once in a while, the edges in the cut. When an edge with a relatively high weight is found, conceptually, the algorithm examines all the edges in the cut. It looks for a new placement that will keep the value of the cut as small as possible. In practice, this examination may be done for only a subset of the edges on the cut or may be done incrementally to all edges in the cut.

Consider a data tree that is navigated without indices. An edge that is closer to the tree root usually has a weight that is bigger or equal to the weight of all its descendant edges (the TPA may affect this property). The reason is that nodes that are closer to the tree root are visited more often than
nodes that are farther away from the root. So, intuitively, the value of the cut roughly measures the number of times we cross the cut and access the second level. We would therefore strive to keep this value as small as possible.

Of course, with indexes, the situation is more complex. Next, we present a greedy algorithm that attempts to minimize the cut value. The heuristic of keeping the cut minimal as well as the greedy heuristic still needs to be verified experimentally.

The algorithm uses a \((m \times n)\) array \((\text{cutWeightsArray})\). Denote the pages which are above the cut as \((a_1, a_2, ..., a_m)\), and those below the cut as \((b_1, b_2, ..., b_n)\). The first (respectively, second) dimension of \(\text{cutWeightsArray} \)
refers to the $a_i$’s (respectively, $b_i$’s). Each element $(i, j)$ of $cutWeightsArray$ is the new value of the cut in case of an exchange in which we move page $a_i$ to the second memory level and page $b_j$ to the first memory level.

The algorithm is presented in Figure 7.4. The function $toMove(e)$ in line 1 triggers the algorithm, it checks if an edge weight is significantly changed, relative to all other tree edges. The function $calcNewCutValue(i, j)$ in line 2.1.2 calculates the new value of the cut in case perform an exchange, that is, we move page $a_i$ to the second level and page $b_j$ to the first level. $movedUpperPages$ (respectively, $movedLowerPages$) in line 2.2 (respectively, 2.3) holds indices of pages from above (respectively, below) the cut that were already moved below (respectively, above) the cut. These pages should no longer be considered for further exchange. The function $findMinValueIndices(cutWeightArray, MUP, MLP)$ in line 2.5.1, finds the indices $(l, m)$ of the minimal element in the $cutWeightArray$, such that page $a_l$ was not already moved to the second level and page $b_m$ was not moved to the first level. ‘$k$’, a system parameter, in line 2.4 is the number of exchanges (between a page from above the cut and page from below the cut) to be performed at each activation of the algorithm. The function $movePageBelow(p)$ (respectively, $movePageAbove(p)$) in line 2.5.2 (respectively, 2.5.3) moves page $p$ from above (respectively, below) to below (respectively, above) the cut. In line 2.5.4 (respectively, 2.5.5), we add $l$ (respectively, $m$), the index of page $a_l$ (respectively, $b_m$), to $movedUpperPages$ (respectively, $movedLowerPages$) as it was already moved below (respec-
tively, above) the cut.

The function $\text{fixCutWeightArray}(l, m)$ in line 2.5.6, fixes line $l$ and column $m$ in $\text{cutWeightArray}$, as it was changed due to the exchange between page $a_l$ (from above the cut) and page $b_m$ (from below the cut).

For a checked edge $e$ in the cut:
1. if ($\text{toMove}(e)$) /*the weight of $e$ has changed significantly so rearrangement of the cut is needed*/
   2.1 $\text{for}(i = 1; i \leq m; i++)$  
      2.1.1 $\text{for}(j = 1; j \leq n; j++)$  
      2.1.2 $\text{cutWeightArray}[i, j] = \text{calcNewCutValue}(i, j)$
   2.2 $\text{movedUpperPages.clear()}$
   2.3 $\text{movedLowerPages.clear()}$
   2.4 $\text{for}(i = 0; i \leq k; i++)$
      2.5.1 $(l, m) = \text{findMinValueIndices}(\text{cutWeightArray},$
         $\text{movedUpperPages, movedLowerPages})$
      2.5.2 $\text{movePageBelow}(l)$
      2.5.3 $\text{movePageAbove}(m)$
      2.5.4 $\text{movedUpperPages.add}(l)$
      2.5.5 $\text{movedLowerPages.add}(m)$
      2.5.6 $\text{fixCutWeightArray}(l, m)$

Figure 7.4: The Two Memory Levels Algorithm

7.2.2 Web site

We can treat a whole web site as a "big tree-structured" document with additional links that allow navigating between nodes that are related by neither parent-child nor sibling relationship. The PIXSAR algorithm can only deal with tree structures augmented with sibling edges, but not with
a general graph. In Figure 7.5 we suggest a simple algorithm that extends PIXSAR and allows coping with more general graphs obtained by adding relatively few edges to a basic underlying tree.

The main idea underlying this algorithm is to incrementally change the structure of the tree on which PIXSAR operates. Initially, we begin operating with a basic tree (without the additional links). As time progresses, the TPA algorithm updates all the edge weights (of the basic tree and of the additional links) according to the changing workload. From time to time, we check the status of the edge weights and exchange an edge with small weight from the basic tree with an edge that has a larger weight, which is one of the additional links. We maintain an "additional edges" hash table ($AEHash$). It contains edges (links) of the website that are currently not part of the augmented tree. This algorithm is run at a certain frequency (system parameter).

Finally, note that the web site algorithm may be used together with the Two Memory Levels algorithm. It may significantly reduce both the time and money needed to operate a big web sites.
For each traversed node $u$:

1. $f = u.father$ /* in the current tree */
2. $o = AEHash.findMaxEdge(u)$ /* this function finds an edge with maximal weight s.t. one of its nodes is $u$ */
3. if ($o$ exists and $o.weight > f.weight$) /* $(u, o)$ edge is more significant than $(u, f)$ */
   3.1 remove edge $(u, v)$ from the basic tree
   3.2 add edge $(u, v)$ to $AEHash$
   3.3 add edge $(u, o)$ to the basic tree
   3.4 remove the $(u, o)$ edge from $AEHash$
   3.5 update all relevant pointers in the tree
   3.6 update all relevant edge weight counters
   3.7 run the reclustering triggering test on $u$ /* as there was a significant change in the basic tree */

Figure 7.5: The Webiste Storing Algorithm
Chapter 8

Conclusions and Future Work

PIXSAR is a novel workload-directed algorithm for incrementally adjusting XML document placement on disk. In the PIXSAR framework, node weights express storage requirements and edge weights represent the likelihood of co-residing in cache. The data clustering problem as a whole is cast as an augmented (with sibling edges) tree partitioning problem.

We constructed a comprehensive experimental data clustering system that includes a disk page layout and a cache management system for storing native XML data. We devised a novel method for 'exporting' the Saxon query processor into our environment. This method allows us to run arbitrary XPath queries on our system. We also devised efficient methods for recording access patterns, express them as edge weights, and efficiently trigger when the file organization becomes deficient. PIXSAR can be used with varying parameters, which influence memory and runtime costs on the one hand, and
the quality of data placement which affects the amount of page faults, on the other hand.

Extensive experimental evaluation, with a simulated disk, demonstrates promising performance of the PIXSAR algorithm - it usually provides nearly 20% (in some cases even more than 40%) improvement over the static placement DFS algorithm in terms of the number of page faults. Experiments with the PIXSAR algorithm operating over a physical disk were also conducted. This extensive experimental evaluation demonstrates that PIXSAR performs well in practice - it usually provides nearly a 40% (in some cases even more than 55%) improvement over the DFS method in terms of the overall run time. All these experiments show that a static, workload independent storage scheme is inferior to one that incrementally adjusts storage layout via the PIXSAR algorithm. So, PIXSAR is highly likely to perform better than any static storage scheme. To summarize, PIXSAR’s flexibility and efficiency makes it a good candidate for use in workload-directed, off-line or online, incremental clustering of XML documents.

The methodology can be easily extended to deal with insert and delete operations. When deletion/insertion takes place, the organization of the document in disk is affected, and reorganization may be necessary. This reorganization may be done immediately or during ”system quite times”.

Deletion causes the creation of ”holes” in some disk pages. To reorganize can immediately, we run PIXSAR on the root node of some cluster that is placed in the ”affected” page. This operation improves the physical organization
of the document. As time progresses, more reclustering operations will be performed in the "affected" area. This will lead to more data placement improvements.

For simplicity, suppose a single element is inserted. An inserted element is usually placed in empty space found in some disk page, or in a new page. Weights on all the edges among a "new" element and "old" elements, are set to the average weight of edges in the current document. After placing the new element, PIXSAR is run, triggered by an edge that connects the new element to the pre-updated portion of the document. This leads to rearranging disk data. As time progresses, the inserted element will probably be traversed, and its associated edges’ weights will eventually express their true weight. Eventually, these edges may participate in reclustering operations and this will lead to further data placement improvements.

The iPIXSAR algorithm, a novel workload directed algorithm for incrementally adjusting XML document placement on disk while taking indices into account was designed and implemented. In this framework, node weights express storage requirements and edge weights represent the benefit of co-residing in cache. The problem as a whole is cast as an augmented (with sibling edges) tree partitioning problem which takes into account indexed nodes. iPIXSAR is able to express affinity between indexed nodes which are not connected via structural edges (child-parent, siblings). This work addresses a new problem that, as far as we know was never investigated, the clustering of multi-rooted tree (MRTs). Practically, we view each index as
an additional root to some subtrees of the basic tree. There is no known $O(nW^k)$ precise partitioning algorithm for this problem (where $n$ is number of nodes, $W$ is the page size limit and $k$ is a constant). This problem is non-trivial and presents a future work direction. In using iPIXSAR, via simulations, performance enhancements are observed over "ordinary" PIXSAR. We have not yet verified this mechanism on a physical disk. iPIXSAR can be used with varying parameters, which influence memory and runtime costs on the one hand, and the quality of data placement which affects the amount of page faults, on the other hand. We plan to further investigate clustering of documents with indices on a real disk and expect to observe significant improvements. We also contemplate various improvements over the version of iPIXSAR uses in the simulations.

A few designs that integrate PIXSAR within traditional disk(s) based computer systems were introduced. Using PIXSAR (or for that matter any incremental reclustering algorithm) within novel architectures was also considered. We examined storing a whole large web site and using remote (and practically unlimited) storage. Algorithms for these settings are presented in section 7.2. Testing the new architectures is planned as future work.

We also intend to investigate the determination of values for various key parameters as well as learning them, autonomously, by the algorithms (self-tuning). Another promising future research direction is designing a light concurrency control mechanism to enable concurrent query processing and reclustering.
Bibliography


Appendix A

Lukes’ Algorithm

XS, which is used by PIXSAR, is based on Lukes’ algorithm. We present a short overview of Lukes’ algorithm. Consider a rooted tree $T = (V, E)$, where $V$ is a set of nodes and $E \subseteq V \times V$ is a set of edges. A cluster over $T$ is a non-empty subset of $V$. When no confusion arises, we simply use the term cluster. A partition of $T$, $P_T$, is a set of pair-wise disjoint clusters over $T$ whose union equals $V$, that is $P_T = \{c_1, \ldots, c_k\}$, $k \geq 1$, such that $\bigcup_{i=1}^{k} c_i = V$, and $c_i \cap c_j = \emptyset$, for all $i \neq j$.

Each node $i$ of $T$ has a weight value (which is the size of the node data), $w_i$. Each edge $(i, j)$ also has a value that represents the importance of the edge, $v_{ij}$. The size of a cluster $c$, is $\text{size}(c) = \Sigma_{i \in c} w_i$. The value of a cluster $c$, is $\text{value}(c) = \Sigma_{(i,j) \in E \land i \in c \land j \in c} v_{ij}$. The value of a partition $P_T$, is $\text{value}(P_T) = \Sigma_{c \in P_T} \text{value}(c)$. The cost of a partition $P_T$ can be defined in two ways: 1. $\text{cost}(P_T)$ is the sum of the values of all edges of $T$ minus...
value($P^T$), namely the total value of inter-cluster edges. 2. cutValue($P^T$), namely the total value of outer-cluster edges. The lower the cost is, the better the partition.

Let $W$, the cluster weight bound, be a positive integer. The tree partitioning problem is formulated as follows: Find a highest value partition, $P^T_{opt}$, among all the possible partitions of $T$, such that the size of no cluster in $P^T_{opt}$ exceeds $W$. $P^T_{opt}$ is said to be an optimal partition. So, $P^T_{opt} = \{c_1, \ldots, c_k\}$ such that size($c_i$) $\leq W$, for $i = 1, \ldots, k$, and value($P^T_{opt}$) = $Max\{value(P^T) | P^T$ is partition of $T$ and $\forall c \in P^T, size(c) \leq W\}$.

Lukes’ algorithm solves the tree partitioning problem. It operates on a tree in a bottom-up manner; it processes a node only after all the node’s children have been processed. Consider a partition $P^{T'}$ of a subtree $T'$ of $T$ rooted at node $x$. The unique cluster in $P^{T'}$ which contains $x$ is called the pivot cluster of $P^{T'}$. Lukes’ uses dynamic programming as follows. For each subtree, say rooted at a node $x$, and for each feasible total cluster size $U$ (i.e., $w_x \leq U \leq W$), it constructs, if possible, an optimal subtree partition in which the pivot cluster is of size $U$. So, Lukes’ algorithm associates a set of partitions with node $x$, each optimal under the constraint that the pivot cluster size is $U$.

When considering node $x$, the algorithm partitions that are associated with each child node of $x$, are used to update the collection of partitions, one per each feasible total cluster size, for $x$. Once the tree root node is processed, the final result, $P^T_{opt}$, is the highest value partition associated with the root; as Lukes showed, $P^T_{opt}$ has the maximum partition value among all possible
partitions of the tree.

To illustrate the Lukes’ algorithm, we consider the XML tree \( T \) displayed in Figure A.1(a), the bold numbers denote nodes. Figure A.1(b) presents the corresponding clustering tree \( T' \) in which for simplicity node weights (inside the circles) are the text sizes of the corresponding XML nodes, and the edge weights (next to the edges) model navigational behavior. Figures A.1(c) and (d) illustrate the clusterings corresponding to cluster weight bounds of 20 and 25, respectively.

Figure A.1: An XML Tree partitioned by using Lukes’ Algorithm.
Appendix B

XML Generator’s DTD

<!-- DTD for recepies database -->

<!ELEMENT collection (description, (meat_recipe | cake_recipe |
pie_recipe | fish_recipe | bread_recipe|
  icecream_recipe)*)>

<!ELEMENT description ANY>

<!ELEMENT meat_recipe (title,(ingredient_complex | ingredient)*,
  preparation,comment?,nutrition)>

<!ELEMENT cake_recipe (title,(ingredient_complex | ingredient)*,
  preparation,comment?,nutrition)>

<!ELEMENT pie_recipe (title,(ingredient_complex | ingredient)*,
  preparation,comment?,nutrition)>

103