MULTIDIMENSIONAL XML FILE: A NEW XML FILE STRUCTURE

BY

MAHBOUB ALI MOHAMMED NAJI

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

JUNE, 2010

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Mahboob Ali Mohammed Naji** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in **COMPUTER SCIENCE**.

Thesis Committee

[illegible]

Thesis advisor

Dr. Salahadin Adam

[illegible]

Member

Dr. Mohammed Almulhem

[illegible]

Member

Dr. Sajjad Mahmood

عادل

Department Chairman

Dr. Adel Fadhl Ahmed

[illegible]

Dean of Graduate Studies

Dr. Salam A. Zummo

24/1/11

Date

عمادة الدراسات العليا

DEANSHIP OF GRADUATE STUDIES

DEDICATION

I dedicate this thesis with all of my love to the prophet of humanity and peace

MOHAMMED RASOOL ALLAH

ACKNOWLEDGMENT

I thank Allah (SWT) for granting me health, patient, guidance and determination to successfully accomplish this work.

Acknowledgment is due to the King Fahd University of Petroleum & Minerals for supporting this research.

I wish to express my deep appreciation to Dr. Salahadin Adam who served as my major advisor for his constant help, guidance, encouragement and invaluable support. I also wish to thank the other members of my thesis committee Dr. Mohammed Al-Mulhem and Dr. Mahmood Sajjad for their cooperation, comments and support.

I would like to thank King Fahd University of Petroleum and Minerals for sponsoring me throughout my graduate studies. I also would like to thank Aden Community College which gives me the opportunity for completing my MSc degree in KFUPM.

Also, I would you like to thank my parents, my wife, my son and daughters who always support me with their love, patience encouragement and constant prayers. I would like to thank my brothers, sisters, and all my friends for their love and support throughout my study.

TABLE OF CONTENETS

LIST OF FIGURES

LIST OF TABLES

THESIS ABSTRACT

NAME: **Mahboub Ali Mohammed Naji**

TITLE: **Multidimensional Xml File: A New Xml File Structure**

MAJOR FIELD: **Computer Science**

DATE OF DEGREE: **June, 2010.**

Exchanging data between organizations becomes challenge because of differences in data formats and in the semantics of the meta-data used to describe the data. E**X**tensible **M**arkup **L**anguage (XML) is playing an increasingly important role in the exchange of a wide variety of data on the Web. Querying XML data is a challenging task because of the nature of XML structure. Unlike flat files, XML documents have nested structure. Querying XML data involves not only the content but also the structure of XML data.

The increasingly wider use of XML has heightened the need to store large volumes of data encoded in XML, and to query XML data more efficiently.

The way an XML document is stored in a secondary storage affects the cost of querying the data. Many techniques of storing XML data have been proposed in the literature. The main disadvantage of the existing techniques is that they organize the data in a way that will result in many disk I/Os to answer a query. In this thesis we are proposing a new multidimensional file structure to store XML data in a secondary storage. This multidimensional file structure will minimize the number of disk blocks accessed to answer a query.

خلاصة الرسالة

الاســــم : محبوب علي محمد ناجي

عنوان الرسالة : ملف اكس ام ال (XML) متعدد الابعاد : ملف جديد لخزن واسترجاع ملفات الاكس ام ال

التخصص : علوم حاســب الي

تاريخ التخرج : يونيــــو 2010

إن تبادل البيانات بين المؤسسات أصبح يشكل تحديا وذلك بسبب الاختلاف في صيغ البيانات إضافة إلى الاختلاف في البيانات الوسيطة أو المساعدة المستخدمة في وصف هذه البيانات المتبادلة. إن لغة الترميز الموسعة الـ (XML) تلعب دورا هاما في تبادل أنواع كثيرة ومختلفة من البيانات خلال صفحة الانترنت. إن الاستعلام عن البيانات الموجودة في ملفات الـ (XML) تعتبر مهمة صعبة وذلك بسبب طبيعة هذه الملفات, حيث وأنها تختلف عن الملفات النصية العادية كون استعلام أو البحث في الملفات النصية العادية يتعلق بمحتوى هذه الملفات ولا يتطلب معرفه تركيب هذه الملفات, بينما البحث أو الاستعلام في ملفات الـ (XML) يتعلق بمعرفة محتوى وتركيب هذه الملفات.

إن الطريقة التي تخزن بها ملفات الـ (XML) في الذواكر الثانوية كالقرص الصلب, تنعكس أو تؤثر في طريقة البحث أو الاستعلام عن البيانات في هذه الملفات.

هناك العديد من الطرق أو التقنيات – في الأبحاث السابقة- استخدمت لتخزين ملفات الـ (XML), ولكن المشكلة الأساسية التي تعاني منها هذه الطرق هي أن الاستعلام عن ملفات الـ (XML) يتطلب قراءه عدد كبير من مقاطع الذاكرة (Disk blocks) وهذا يعني عمليات عديدة من عمليات القراءة والكتابة من والى الذاكرة الثانوية للإجابة عن هذا الاستعلام.

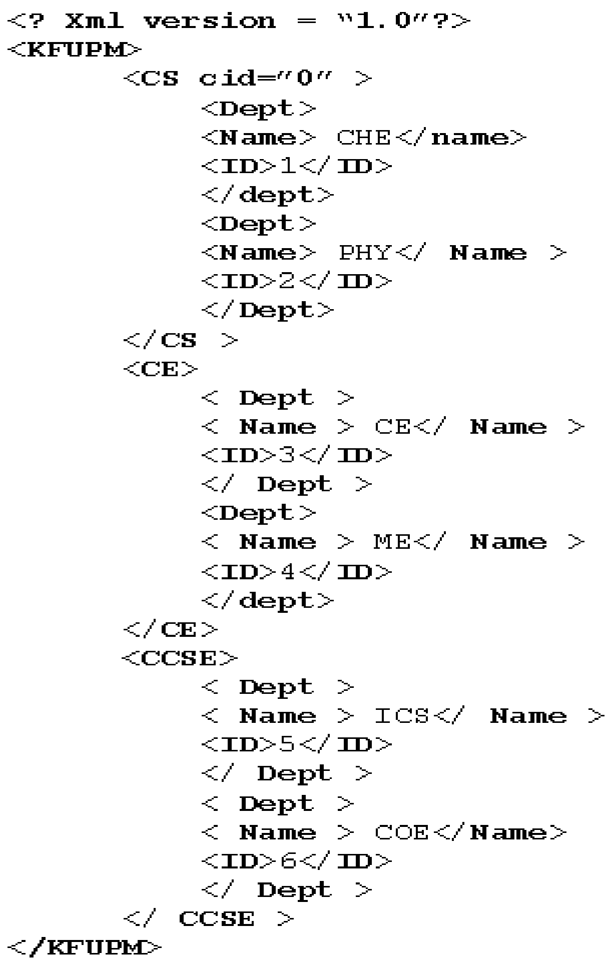
في هذه الأطروحة نقدم أو نقترح طريقة جديدة لتخزين ملفات الـ (XML) في الذاكرة الثانوية, هذه الطريقة تخزن ملفات الـ (XML) بشكل ملف متعدد الأبعاد(Multidimensional structure file) وفكرة هذه الطريقة أنها تخزن البيانات بطريقة تجعل عدد مقاطع الذاكرة المطلوب الوصول إليها قليل عند الاستعلام عن بيانات في ملفات الـ (XML).

CHAPTER ONE

INTRODUCTION

1.1. Background

Data exchange among organizations is challenging because of differences in data formats and in the semantics of the meta-data used to describe the data. E**X**tensible **M**arkup **L**anguage (XML), a simple and very flexible text format, is playing an increasingly important role in the exchange of a wide variety of data on the Web. The simplicity, flexibility, and data self-describing capability of XML, makes it ideal for data exchange [1] [2]. XML data is self describing because XML tags are used to describe the semantic of the data. For example, in Figure 1.1, the tag < name> nested in tag < Dept> means the name of the dept. XML is flexible in organizing data, that is, objects of the same type might have different types of sub objects or different numbers of sub objects of the same type [3]. Since of the importance of XML as a new standard for information representation and exchange on the Internet, the problem of storing, indexing, and querying XML documents poses new challenges to database researchers, and has been among the major issues of database research [4]. XML documents form a tree structure that starts at "the root" and branches to "the leaves". So an XML document can be represented as a tree, where nodes represent the element of the document and edges represent the relationships between these elements [5].



```
<? Xml version = "1.0"?>
<KFUPM>
      <CS cid="0" >
          <Dept>
          <Name> CHE</name>
          <ID>1</ID>
          </dept>
          <Dept>
          <Name> PHY</ Name >
          <ID>2</ID>
          </Dept>
      </CS >
      <CE>
          < Dept >
          < Name > CE</ Name >
          <ID>3</ID>
          </ Dept >
          <Dept>
          < Name > ME</ Name >
          <ID>4</ID>
          </dept>
      </CE>
      <CCSE>
          < Dept >
          < Name > ICS</ Name >
          <ID>5</ID>
          </ Dept >
          < Dept >
          < Name > COE</Name>
          <ID>6</ID>
          </ Dept >
      </ CCSE >
</KFUPM>
```

Figure 1.1: a simple XML document

XML document in Figure1.1 can be represented as a tree showed in the Figure 1.2

1.2. Numbering scheme

XML data can be represented as a rooted, ordered and labelled tree. In Figure 1.2 XML document represented as tree, where a node is either (1) *an element node* (internal node) which corresponds to a tag in XML document, such as *dept*, *name*, etc. (2) *An attribute node* (internal node) which corresponds to an attribute node such as *cid*. (3) *A value node* (leaf node) which corresponds to data value such as *CHE*, *1*, etc. Edges indicate the relation between nodes which is either parent- child relation or ancestor- descendant relation. Parent- child relation (direct containment) such as the relation among *KFUPM* and *CS* nodes and ancestor- descendant relation (indirect containment) such as the relation among *KFUPM* and *Dept* nodes.

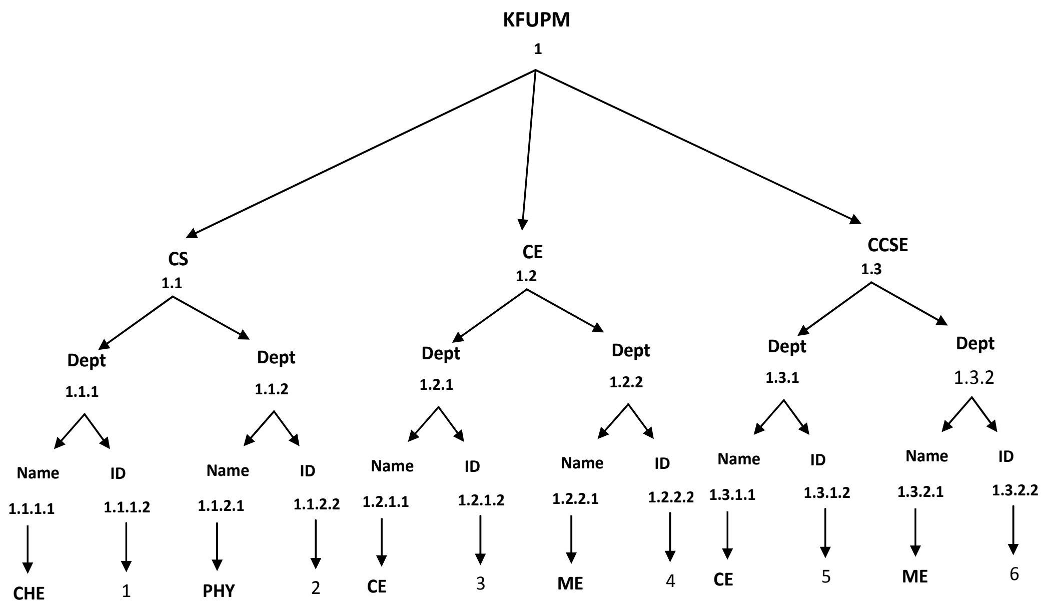


Figure 1.2: a tree representation of the XML document of Figure1.1

1.3. Storing XML data

The cost of a query can be measured mainly by the number of disk I/Os performed to answer the query. As a result the way XML data is organized in a secondary storage affects the cost of a query significantly.

Recently, there has been a lot of interest in XML data storage strategies. Existing XML storage strategies can be classified into three main approaches, namely, *file approach, relational approach, and native approach.*

1.3.1. File approach

In this approach XML document is stored as a separate operating system file. A DOM parser is used whenever the document is accessed by a query. This approach is trivial to implement and DOM parsers are widely available [6].

1.3.2. Relational approach

Another approach to store XML data is *the relational approach* in which XML data are stored in relational databases.

The main advantage of this approach is that existing important techniques can be reused and no need for extra development efforts. Queries written in XML query languages, such as XML-QL, are translated in to SQL and executed by the underlying relational database system [7] [8] [9] [10] [11] [12] [13].

But this in turn, leads to the following problems that need to be addressed in order to store and query XML data in relational databases:

- *Schema mapping*, to generate the relational schema from a DTD.
- *Data mapping*, to inserts XML data as relational records into the target database.
- *Query mapping*, to translate XML queries into SQL queries.
- *Reverse data mapping*, to publishes XML data from relational data

*A*ll these mappings are very costly in both time and space because there is a need to perform multiple joins between tables [14] [15].

1.3.3. Native approaches

There are several native approaches. In one native approach, XML data are stored in inverted lists and native query processing algorithms are developed to process them. The concept of inverted lists originated from inverted indexes, which have been widely used in information retrieval to search for text efficiently. An inverted list is created for each distinct tag in XML documents. Each list records the positions of all elements with that tag name, where the position of an element is expressed using its (Start, End, Level) numbers (or Dewey vectors). Elements in each list are sorted in the increasing order of their start numbers [16] [17] [18]. Figure 1.3 below shows an example of inverted lists of the elements *Dept* and *Name* in the Figure 1.2.

Dept	1.1.1	1.1.2	1.2.1	1.2.2	1.3.1	1.3.2
Name	1.1.1.1	1.1.2.1	1.2.1.1	1.2.2.1	1.3.1.1	1.3.2.1

Figure 1.3: inverted lists for Dept and Name elements

Native XML database is a database that has an XML document as its fundamental unit of (logical) storage and defines a (logical) model for an XML document. It represents logical XML document model, and stores and manipulates documents according to that model.

The main features of a Native XML database are the following:

- XML document or its rooted part is a logical unit of a Native XML database.
- At least the following components are included: elements, attributes and textual data.
- Physical model is unspecified, which implies that XML documents storage may be arbitrary, as long as it stores and manipulates an XML document as a (logical) unit.

Native XML databases are especially suitable for storing irregular, deeply hierarchical, recursive data.

The advantages of using native XML databases over other types of databases are numerous. They free users from having to know document schema, they support data models that does not fit other databases (e.g., relational databases), and provide for extensibility, etc [16] [17] [18].

1.4. Multidimensional file structure

There are many file structures available for managing a collection of records identified by a single key like: sequentially allocated files, tree-structured files, and hash files.

Because of the increasing usage of databases and integrated information systems, there is a real need now for file structures that allow efficient access to records by combinations of attribute values. This is what is called multi-key access. These file structures have to be efficient, also in a highly dynamic environment, i.e. when there is a high rate of insertions and deletions.

1.5. Thesis contribution

Previous native XML storage systems depend on the inverted lists to store the file. They save the inverted lists of all elements in an XML documents, by decomposing paths of the XML tree and save the elements of the paths separately (using their numbering scheme).

There are two major disadvantages of these approaches: first, it uses large disk space, and second it needs many join operations to process a query. This thesis proposes a new file structure Multidimensional XML File (MXF) to save, index and query XML document. The main idea of our proposed system is to store the inverted lists of the leaf nods only. The proposed approach uses less disk space, minimizes the need for join operations in querying twig queries, and eliminate join operations in simple path queries.

The main thesis goals will be as follow:

- To find a new multidimensional file structure to store XML data.
- To implement the new multidimensional file structure.
- To find a suitable directory structure for the multidimensional file structure.
- To implement the new directory structure.
- To find the density of the new multidimensional file structure and the density of its directory.
- To study how the multidimensional file structure is affected by:
    - Shape of an XML tree
    - Levels of an XML tree
    - Number of elements in an XML tree
    - Size of an XML tree
- To implement delete, update, and select operations on the new multidimensional file structure.
- To find a suitable node labeling scheme if needed.

1.6. Thesis outlines

The rest of this thesis will be organized as follows:

- Chapter two presents the literature review. We discuss the previous XML storage approaches, the existing XML numbering scheme, the XML query processing approaches, and the multidimensional files.
- Chapter three shows the proposed MXF system. We explain in details the structure of MXF and we show by examples how MXF works to index and save an XML document. Finally, we discuss the main MXF operations: searching, insertion, and deletion.
- Chapter four introduces the experimental results and analysis. We apply our MXF on different XML databases and study its behavior. Environment, datasets, query sets and performance metrics are discussed. We use different data sets according to their sizes, their distinct paths, and their shapes (how deep or wide an XML document is).
- Chapter five concludes this thesis. Also, describes a number of future works.

CHAPTER TWO

LITERATURE REVIEW

The increasingly wider use of XML has heightened the need to index and store large volumes of data encoded in XML, and to query XML data more efficiently. Many XML storage approaches, XML numbering schemes and many XML query processing algorithms have been proposed in literature. In this chapter we briefly introduce some of them.

The rest of this chapter will be organized as follows. Section 2.1 introduces the existing numbering schemes. Section 2.2 shows the existing XML storage approaches. Section 2.3 explains the XML query processing approaches. And finally, section 2.4 presents multidimensional files.

2.1. The XML Numbering Scheme

Querying XML documents can be done by a combination of value search and structure search. In the value search, a query is processed by matching specified values or by matching specified element or attribute names. Searching by structure depends on the element to element or element to attribute relationship.

In order to process the relationship between nodes, the nodes in an XML tree are systematically labelled in such a way that the structural relationship (e.g., the ancestor–descendant relationship) between two arbitrary nodes can be computed efficiently.

Various labeling techniques are proposed in literature. The following are some of the main labeling approaches: the navigation approaches, the prefix-based approaches and prime number approach.

2.1.1. The navigation approach

The *navigation approach*: a straightforward method for determining reachability is tree navigation which either traversing down (forward navigation) or backtracking (backward navigation) an XML tree [19] [20]. However, the navigation method is, in general, not very efficient, since it may involve traversing a large number of query-irrelevant nodes.

An interval (region) approaches: in this method the reachability between two nodes could be determined through checking the containment relationship between their intervals.

Dietz in [21] proposed the original work on numbering schemes for trees. He proposed PrePost numbering scheme. PrePost labels each node in a tree with a pair of numbers: (pre, post), which correspond to the pre order and post order traversal numbers of the node in the tree.

Zhang et al. [22] introduced PrePost coding into XML: it labels each node in an XML data tree with a pair of numbers (start, end) which means the position of the opening tag and the closing tag of the corresponding element of the node in the XML tree. Figure 2.1 shows the interval numbering scheme.

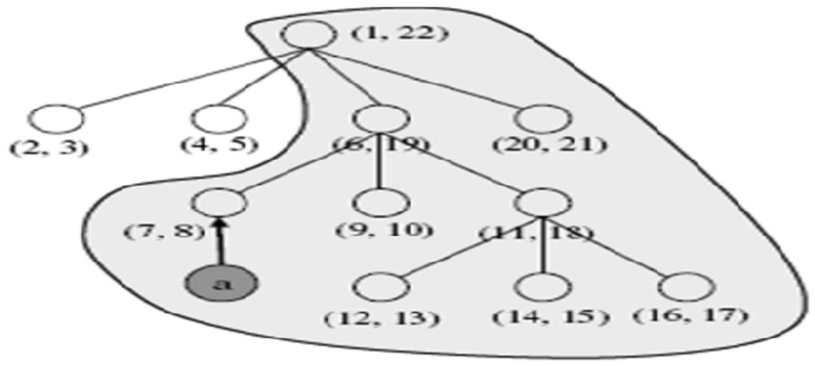


Figure 2.1: region based numbering scheme

2.1.2. Prefix- based approach

The *Prefix- based approach*: in this scheme each node of an XML tree has a string label which is the concatenation of the parent's label and its own identifier (i.e., self-label). If there are two nodes x and y where x is an ancestor of y, then label(x) is a prefix of label(y). Dewey labeling scheme [23] [24] and Binary labeling [25] are two examples of prefix based labeling scheme.

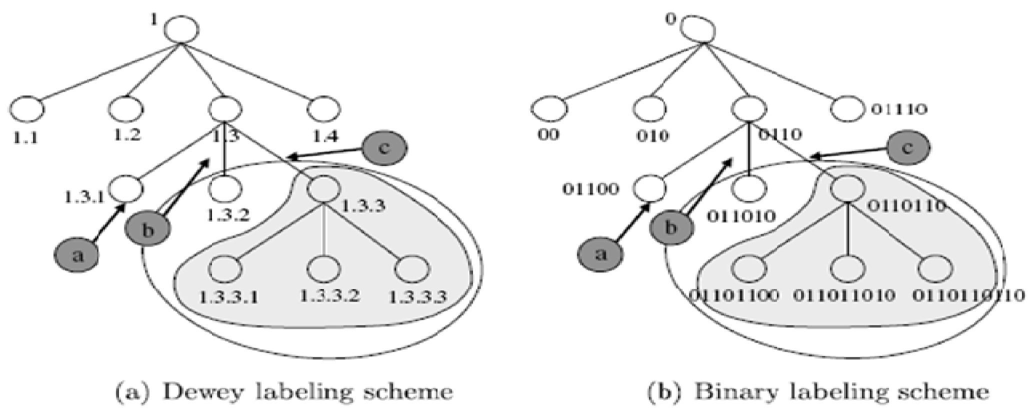
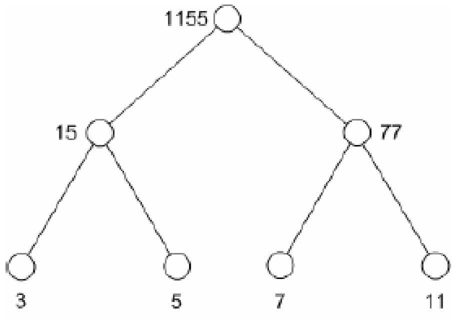


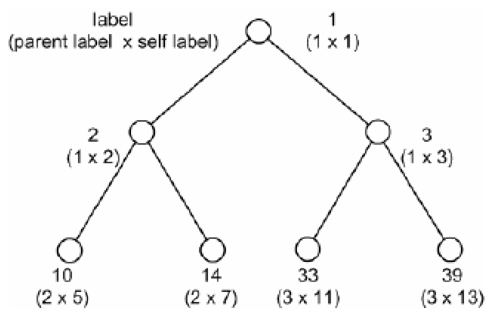
Figure 2.2: prefix based labeling scheme

2.1.3. The prime number approach

The prime number approach: where each node is labeled by an integer and the labeling scheme ensures that each label can only be divided exactly by its own ancestor in an XML tree [26]. There are two types of this scheme bottom-up and top-down prime number labeling scheme. In the bottom-up prime number labeling scheme, for any nodes x and y in an XML tree, x is an ancestor of y if and only if label(x) mod label(y) = 0 as shown in figure 2.3 (a). In the top-down approach the label of a node x equals to its label multiplied by the label of its parent as shown in Figure 2.3(b).



a



b

Figure 2.3: prime number labeling scheme

2.2. XML Storage Approaches

Recently, there has been a lot of interest in XML data storage strategies. Existing XML storage strategies can be classified into three main approaches, namely, *file approach, relational approach, and native approach.*

2.2.1. File approaches

XML data are originally created in the form of XML documents and stored in flat files. Generally, various indexes need to be built on XML data to facilitate answering XML queries.

This approach is easy to implement and does not require the use of a database system or storage manager. But, this strategy has several major disadvantages. First, the whole

XML file must to be parsed to retrieve the data specified by a query. Second, the whole parsed file, which is always much larger than the original XML document, has to be stored in memory during query processing. Third, building and maintaining indices on flat documents is hard. Finally, indexes themselves are huge.

M, Altınel, et.al in [27] have developed several index organizations and search algorithms for performing efficient filtering of XML documents for large-scale information dissemination systems.

N. Bruno et.al in [28] introduced a new index based technique, Index-Filter, to answer multiple XML path queries. Index-Filter uses indexes built over the document tags to avoid processing large portions of the input document that are guaranteed not to be part of any match.

Y Diao et. al in [29] have developed YFilter, an XML filtering system that provides fast, on-the-fly matching of XML encoded data to large numbers of query specifications containing constraints on both structure and content.

2.2.2. Relational approaches

Many relational approaches have been proposed and they are either schema-aware or schema-oblivious approaches.

2.2.2.1. Edge-approach

Edge approach is one of the schema-oblivious approaches. In this approach all edges in a data tree are stored in a single relational table called Edge. The schema of this Edge table is (*label, source, target, flag, and value*). The key idea of this schema is an attribute pair (*Source, Target*), which represents end points of edges. *Label* attribute represents tags on edges. *Flag* and *Value* attributes give the type and value of target nodes of edges, respectively. One of the edge-approach that was proposed by Florescu and Kossmann [30] was shredding XML data into relations. This approach places all edges in an edge-labeled XML data tree into a single relational table called Edge table and the schema of this table is(*label, source, target, flag, and value*).

2.2.2.2. The node approach

The *node approach* (one of the schema- oblivious approaches), that is similar to the edge approach, in which all internal nodes (that is, the element and attribute nodes) in a node-labeled XML data tree are stored in a relational table.

Zhang et al. [22] proposed the *node approach* that is similar to the edge approach, in which all internal nodes (that is, the element and attribute nodes) in a node-labeled XML data tree are stored in a relational table.

2.2.2.3. The DTD approach

The *DTD approach* is one of the schema-aware approaches, stores XML document using its associated DTD. DTD is a set of statements that describe the objects and their relationships that are allowed in an XML document. DTD can be mapped into relational schemas. The number of relational tables created depends on the relationship of the objects specified in the DTD.

2.2.2.4. The Path based approach

In a Path based approach, node-labeled XML data tree is stored in a relational table called Path. The Path table is very similar to the Node table. The difference is that rather than storing the tag of each node, the path approach stores the tag path from the root to each node. Yoshikawa et al. [31] proposed a Path Materialization (PM) approach, in which internal nodes in a node-labeled XML data tree are stored in a relational table.

Pal et al. [32] proposed a Reversed-Path (RP) approach. The RP approach uses the relation schema. The key idea of RP is storing reversed root paths of data nodes in a Reversed Path attribute.

2.2.3. Native approaches

These approaches have been developed to overcome the disadvantages of relational approaches.

University of Mannheim's database research group in [33] proposed *Natix* system which divides XML documents into sub trees according to the physical disk page size (each sub tree is a record).

TIMBER in [34], implemented in University of Michigan, transforms the XML document into a parse tree, which it stores as an atomic unit in the underlying storage manager. The system is based upon a bulk algebra for manipulating trees, and natively stores XML

Ipedo in [35] proposed by IPEDO, maintains, on disk, the physical data files that store collections and metadata associated with collections.

Apache Xindice community in [36] have proposed *Xindice*: stores the entire XML document as a single record (Document-based storage).

eXist, founded by Wolfgang Meier in [37], is an open source database management system built using XML technology. It stores XML data according to the XML data model and features efficient, index-based XQuery processing. eXist stores documents either in the internal XML store or on an external relational net database.

K. Staken in [38] introduces the *dbXML*. *dbXML, an* open source native XML database, offers either XML document-based storage or binary streams (records)

Quanzhong Li et al. in [39] proposed an XML Indexing and Storage System (XISS) as a native XML indexing and storage on a new numbering scheme for elements and attributes.

XISS is composed of three major components: *element index*, *attribute index* and structure index. Name index maps element or attribute name to a name identifier then all distinct name strings are collected in the name index, which is implemented as a B+-tree. *Attribute index* is like the element index but it stores attributes instead of elements.

In the *structure index* the input is the document identifier (*did*) and the output is an array containing all the element and attributes in the document. It is also implemented by a B^+-tree.

The main idea of XISS is to decompose the elements in the documents and store them separately.

N. Zhang, V. Kacholia, and M. Tamer Ozsu in [40] propose a succinct physical storage scheme as a native XML data storage. The idea of this scheme is to store structural information separately from value information. XML data tree should be "materialized" to fit into the paged I/O model. Materialization means the two-dimensional tree structure should be represented by a one-dimensional "string".

After the separation, connecting between structural information and value information is needed. Dewey ID is used to reconnect the two parts of information.

2.3. XML Query processing

Querying XML document effectively and efficiently is still a challenging issue. Unlike keyword search in text retrieval, which concerns only contents of text documents, XML queries concern structure as well as contents of XML documents. In this section we will introduce the most famous XML query languages as well as the XML query processing approaches.

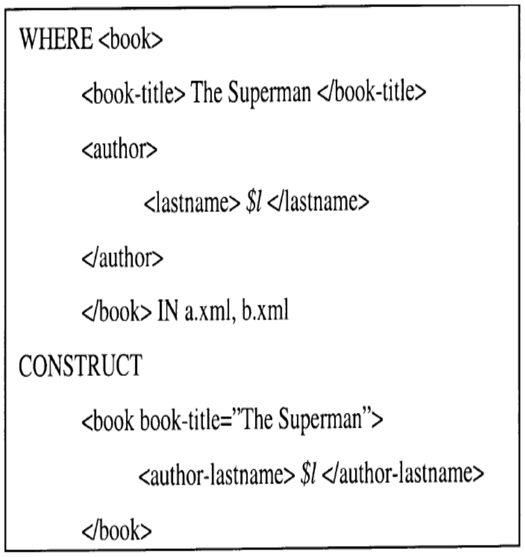
2.3.1. Xml Query Languages

There are many query languages that can be used to query XML document, Lorel, XML-QL (XML-Query Language), XML-GL (XML-Graphic Language), XPath, and XQuery. The core part of an XML query language is the path expression notation for navigating the XML nested structure. XML-QL and XQuery are proposed by W3C as standard XML query language.

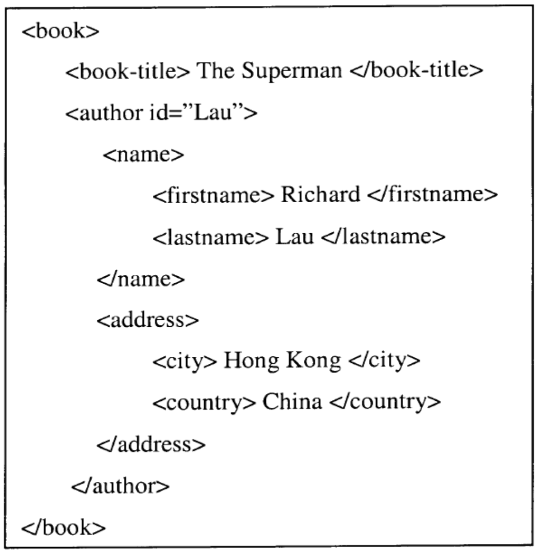
2.3.1.1. XML-QL

XML-QL is the first XML query language proposed by W3C.It uses a nested XML-like structure to specify the part of document to be selected and the structure of result of XML document.

Figure 2.4 shows an XML-QL example for a simple XML document in Figure 2.5.



```
WHERE <book>
        <book-title> The Superman </book-title>
        <author>
                <lastname> $l </lastname>
        </author>
        </book> IN a.xml, b.xml
CONSTRUCT
        <book book-title="The Superman">
                <author-lastname> $l </author-lastname>
        </book>
```



```
<book>
     <book-title> The Superman </book-title>
     <author id="Lau">
          <name>
               <firstname> Richard </firstname>
               <lastname> Lau </lastname>
          </name>
          <address>
               <city> Hong Kong </city>
               <country> China </country>
          </address>
     </author>
</book>
```

Figure 2.4: an XML-QL example **Figure 2.5: a simple XML document**

The query above is to determine the last-name of the author of a book having title" The Superman" from a.xml and b.xml documents. The retrieved value of the last-name will be put in the $l variable. Then the output will be formatted according to the template specified after CONSTRUCT keyword.

2.3.1.2. XPath

XPath is a basic XML query language that selects nodes from XML documents such that the path from the root to each selected node satisfies a specified pattern. A simple XPath query is formulated as a sequence of alternating axes and tags. Two most commonly used axes are the child axis "/," where "A/B" denotes selecting B-tagged child nodes of A-tagged nodes, and the descendant axis "//," where "A//B" denotes selecting B-tagged descendant nodes of A-tagged nodes. Consider an example: An

XPath query “/book//book-title” would return all book-title elements under all top-level book element.

2.3.1.3. XQuery

XQuery is to XML what SQL is to database tables. XQuery. Query language XQuery is more expressive than XPath. An XQuery query is composed of For-Let-Where-Return (FLWR) clauses, which can be nested and composed with full generality [17]; that is, each clause in itself can include sub–XQuery queries. The For and Let clauses bind nodes selected by XPath expressions to user defined node variables. The Where clauses specify selection or join predicates on node variables. The Return clauses operate on node variables to format query results in the XML format.

2.3.2. XML Query Processing Approach

In this section we will explain the most important two approaches of xml query processing techniques: relational approaches and native approaches.

2.3.2.1. The Relational Approach

Through years, RDBMSs have acquired strong capabilities in storage management, query processing and optimization, and concurrency control and recovery. Motivated by this fact, a number of research projects have addressed storing and querying XML data in RDBMSs.

2.3.2.1.1. The Edge Approach

Florescu et al. In [30] proposed a simple approach to shredding XML data into relations. This approach places all edges in an edge-labeled XML data tree into a single relational table Edge. The key idea here is using an attribute pair (Source; Target), which represents the two end points of each edge. Label represents the tag on an edge, whereas Flag and Value give the type and value, respectively, of the target node of an edge. Two edges A and B can be joined together if and only if A.*Target* = B.*Source*. Based on this property, it is easy to transform XML queries without "//" axes into SQL queries. Evaluation of such SQL queries comprises two main steps. The first step is edge selection, which retrieves the data edges for each label in the query. The second step is edge joining, which joins adjacent data edges retrieved in the first step. This step can be done in a more efficient way by using prebuilt indexes on (Source; Target).

This approach may fail to process queries with "//" axes efficiently and it may involve a number of join operations.

2.3.2.1.2. The Node Approach

Zhang et al. [22] developed a Node approach, in which all internal nodes in a node-labeled XML data tree are stored in a relational table Node. The key idea here is using the attribute triple (Start, End, Level). "//"-axis queries can be answered efficiently by using the (Start, End) pairs. Level is used along with (Start, End) to answer "/"-axis queries. Queries with both "/" and "//" axes are translated into SQL queries. Then it is similar to the Edge approach, evaluation of the SQL queries consists of two steps: node selection and node joining.

The Node approach does support “//”-axis queries efficiently. However, similar to the Edge approach, it may involve a number of join operations.

2.3.2.1.3. The Path Materialization (PM) Approach

To reduce the number of node joins, Yoshikawa et al. [31] proposed a PM approach, in which internal nodes in a node-labeled XML data tree are stored in a relational table Path. Using the Path attribute, the PM approach can answer twig queries efficiently in units of paths rather than in units of edges.

Given a twig query, the PM approach first decomposes it into multiple root-to-leaf path queries and then joins the results of the path queries. Evaluation of the SQL queries consists of two main steps: path selection (part 1) and path joining (part 2).

PM may not support efficiently queries with multiple “//” axes. Another limitation of PM is that it might result in incorrect query answers.

2.3.2.1.4. The DTD approach

The DTD approach transforms XML queries into SQL queries based on the schema information in the DTD tree: For a “/”-axis join A/B, it first checks whether A is the parent of B in the DTD tree. If not, then A/B is an invalid query. Otherwise, relations A and B are joined using

A.id = B.parent-id. For a “//”-axis join A//B, it first checks whether A is an ancestor of B in the DTD tree. If not, then A//B is an invalid query. Otherwise, relations A and B and all relations between them (which can be found in the DTD tree) are joined using the “/”-axis join above.

We can summarize that when XML data are schema-less, the PM approach has advantages over the Edge and Node approaches because PM supports “//”-axis queries efficiently and may require fewer join operations. Also, when XML data conform to a schema, the DTD approach could generally have better performance than other schema-less approaches.

2.3.2.2. The Native Approach

Although the relational approach is simple and straightforward to implement, it may not exhibit optimal query processing performance. Motivated by this, many native techniques have been developed to query XML data efficiently. These techniques are called native approaches, since their query processing (and, perhaps, also storage) mechanisms are developed from scratch, without involving relational databases.

In next sections, we review the Join approaches, very important native approaches, which implement efficiently structural joins involved in XML twig queries. In these approaches, XML data are stored in inverted lists. An inverted list is created for each distinct tag in XML documents, and each list records the positions of all elements with that tag name, where the position of an element is expressed using its (Start, End, Level) numbers.

2.3.2.2.1. The Multi-Predicate MerGe JoiN (MPMGJN) Approach

Zhang et al. [22] proposed an MPMGJN algorithm, whose implementation is somewhat similar to the classical merge join algorithm developed in relational query optimizers for equijoins. To answer a query "A//B" or "A/B," first, two cursors are created to point to the heads of list A and list B, respectively. Then, the two cursors are compared with each other and are advanced as needed to implement the merge join. Figure 2.3 b shows this approach.

2.3.2.2.2. The StackTree Approach

Al-Khalifa et al. [41] observed that MPMGJN fails to process "/"-axis queries efficiently in some cases. She proposed a StackTree approach that can avoid such visiting of unnecessary nodes. StackTree uses a stack structure to cache those A nodes that are nested on the same path in data trees. At each step, the data node with the smallest start number is taken out of its list. If it is an A-tagged node, then it is pushed into the stack. If it is a B-tagged node, then StackTree tries to use it to form tuple solutions with A-tagged nodes in the current stack. Fig. 2.6 c illustrates this process, in which b3 is compared with a3 only (step 6) rather than with a1 through a3, as in Fig. 2.6

b. Generally, StackTree shows better query processing performance than MPMGJN [41]

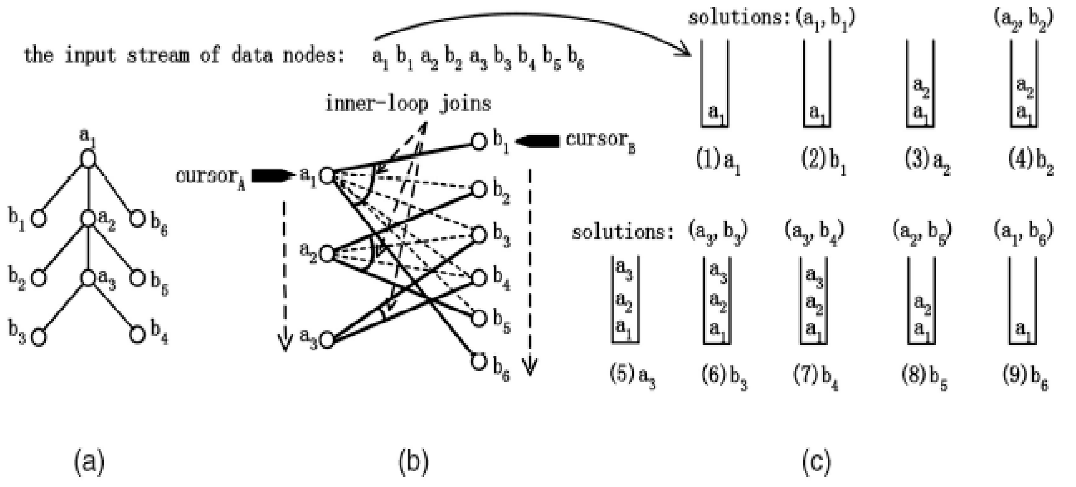


Figure 2.6: applying MPMGJN and StackTree to query "A/B." (a) Data tree. (b) The MPMGJN approach. (c) The StackTree approach.

2.3.2.2.3. The Holistic Approach

StackTree and MPMGJN have to decompose twig queries into multiple binary joins, which might generate a large volume of intermediate query results. This may result in high disk I/O costs Motivated by this observation, Bruno et al. [42] proposed a Holistic approach, whose key idea is pipelining, that is, joining multiple inverted lists at a time to avoid generating intermediate join results.

2.3.2.2.4. The PathStack approach

The Holistic approach to answering (linear) path queries is a PathStack algorithm. The framework of the algorithm is somewhat similar to that of StackTree. The difference is that StackTree uses only one stack to cache nested A nodes. In contrast, PathStack has multiple stacks, one for each node in a path query. In addition, each data node cached in a stack has an associated pointer to a corresponding node in its parent stack in order to track tuple solutions. For an illustration, see Fig. 2.7.

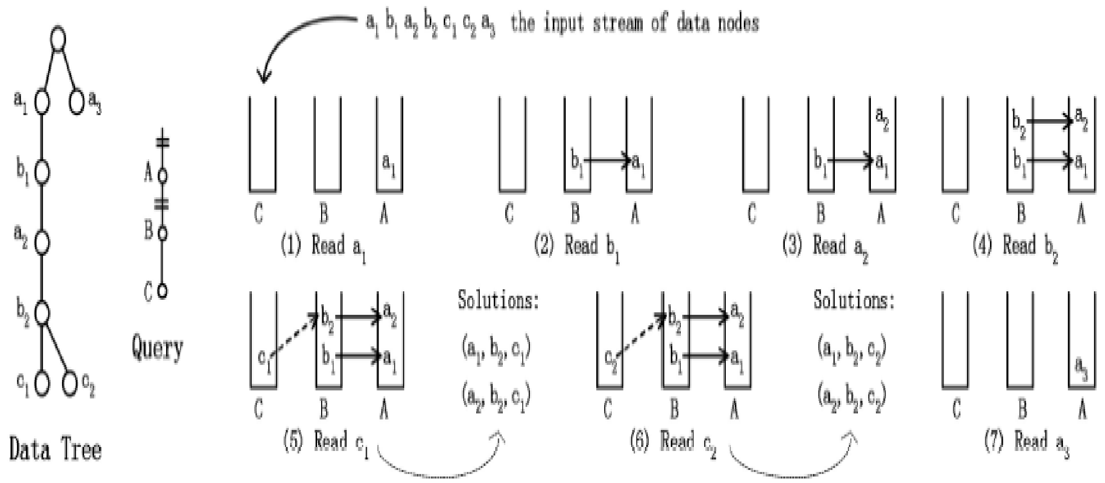


Figure 2.7 an example of the PathStack approach.

2.3.2.2.5. The TwigStack approach

The Holistic approach to answering general twig queries is a TwigStack algorithm [42], which includes two steps: 1) deriving path solutions (that is, the twig query is decomposed into multiple root-to-leaf path queries, and the solutions to these path queries are then derived from the data tree) and 2) joining path solutions.

A simple method for implementing step 1 is to process each path query separately by using PathStack.To reduce the number of such redundant path solutions, the TwigStack algorithm introduced an additional function q = getNext(). Figure 2.8 explains this approach.

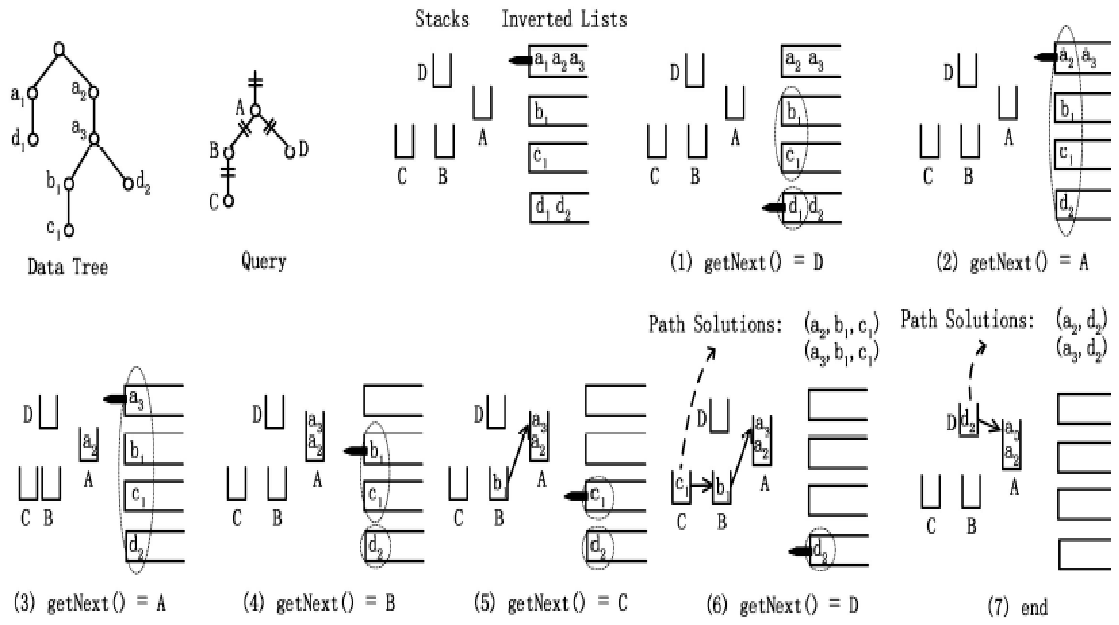


Figure 2.8 an example of the TwigStack approach.

2.4. Multidimensional file structure

Since of the increasing usage of databases and integrated information systems, there is a real need now for file structures that allow efficient access to records by combinations of attribute values.

There are important criteria to assess a multi-key file structure like: the adaptability in a dynamic environment, space utilization, operation speed and the retrieval time. Retrieve time usually measured by a number of disk access to bring data to memory.   Therefore, number of disk accesses is used as the main measure of the efficiency of multi-key file structure. There are many multi-key file structures like in literature; the grid file is one example of multi-key access file structure [43].

2.4.1. The Grid File

The grid file is an adaptable, symmetric, multi-key file structure [44]. Adaptable means it adapts to its contents under modifications like: deletions and insertions. It is a highly dynamic file. The access time is uniform over the entire file, and a single record is retrieved in at most two disk accesses. The grid file is symmetric, as it treats all keys symmetrically. That means it avoids distinction between primary and secondary keys.

The grid file consists of two main parts: the grid file (partition) and the grid directory.

2.4.1.1. The grid partition

The grid file a record is characterized by a number of attributes. Thus, we have a linearly ordered attribute space, and a k-dimensional key to retrieve a record is represented by a point in this attribute space. The attribute space can be represented by a bitmap. In a k-dimensional bitmap, the combinations of all possible values of k attributes are represented by a bit position in a k-dimensional matrix.

2.4.1.2. The grid directory

The grid directory has the function of a bucket management system, which is superposed onto the grid partitions. It consists of two parts as shown in Figure 2.9:

- A dynamic k-dimensional array called grid array. Its elements are pointers to the data buckets and are in one-to-one correspondence with the grid blocks of the record space.

- K one-dimensional arrays called linear scales; each scale defines a partition of a domain.

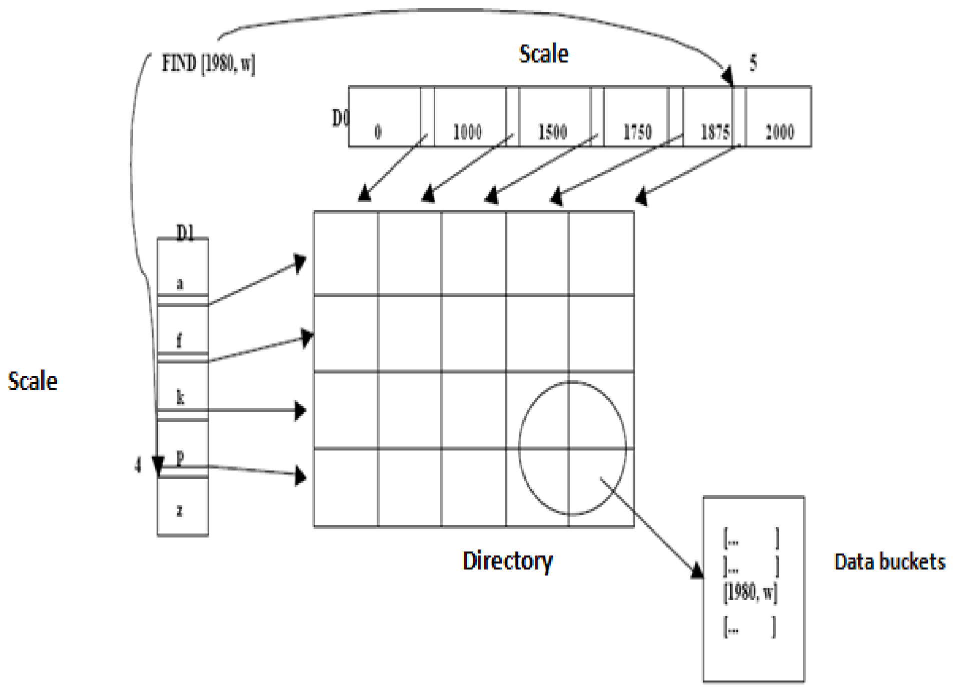


Figure 2.9: a grid file structure

CHAPTER THREE

MULTIDIMENSIONAL XML FILE (MXF)

3.1. Introduction

MXF is a file which indexes and stores XML data in multiple dimensions where XML tree level is considered as a dimension. The benefit of storing XML data in multidimensional way is to make accessing this data easy and accessing them throughout more than one key. MXF extracts all XML document paths, indexes and stores them in a multidimensional way, so MXF is a set of **directory blocks** (blocks contain indexes to access data blocks) and **data blocks** (blocks contain data) as we will see in this chapter.

The rest of this chapter will be organized as follows: section 3.2 explains the structure of MXF that is, MXF parser and MXF indexer. Section 3.3 discusses main operation on MXF (searching, insertion and deletion).

3.2. MXF structure

Functionally, MXF consists of two main parts namely MXF parser and MXF indexer:

3.2.1. MXF parser

Parsing an XML document is the first step before saving it. The idea of parsing is to read XML data from tree form to more organized form. Many parsers exist in the literature and each has its own output according to the information needed to extract from the XML document.

MXF parser is a new parser we have developed to extract information for all paths in an XML documents. The main operations of MXF parser are:

1. Extract all distinct elements form an XML document.
2. Assign each distinct element distinct binary number.
3. Save each element with its corresponding distinct binary number in the tag table.
4. Extract each path in the document and create its binary corresponding path.
5. Generate DEWEY for each path.
6. Generate DEWEY for each attribute if exist.

Let us take this simple example to explain how MXF parser works.

Example 3.1: suppose we have this simple XML document

```
<Library>
        <Main_library>
             <Book>
                   <Title> Database</ Title >
                   <Author> John</ Author >
                   <Year> 2001</ Year >
                   <Price> 20$</ Price >
             </Book>
             <Book>
                   <Title> Operating system</ Title >
                   <Author> Philip</ Author >
                   <Year> 2007</ Year >
                   <Price> 60$</ Price >
             </Book>
             <Magazine>
                   <Title> CPUs structure</ Title >
                   <Year> 2003</ Year >
                   <Price> 30$</ Price >
                   </ Magazine >
             <Magazine>
                   <Title> Today technology</ Title >
                   <Year> 2008</ Year >
                   <Price> 40$</ Price >
             </ Magazine >
        </Main_library>
        <Bookstore>
             <Journal>
                   <Title> 3G mobiles</ Title >
                   <Year> 2008</ Year >
                   <Price> 44$</ Price >
             </ Journal >
        </Bookstore>
</ Library >
```

Figure 3.1: a simple XML document

The above XML document can be represent as a tree like the figure bellow:

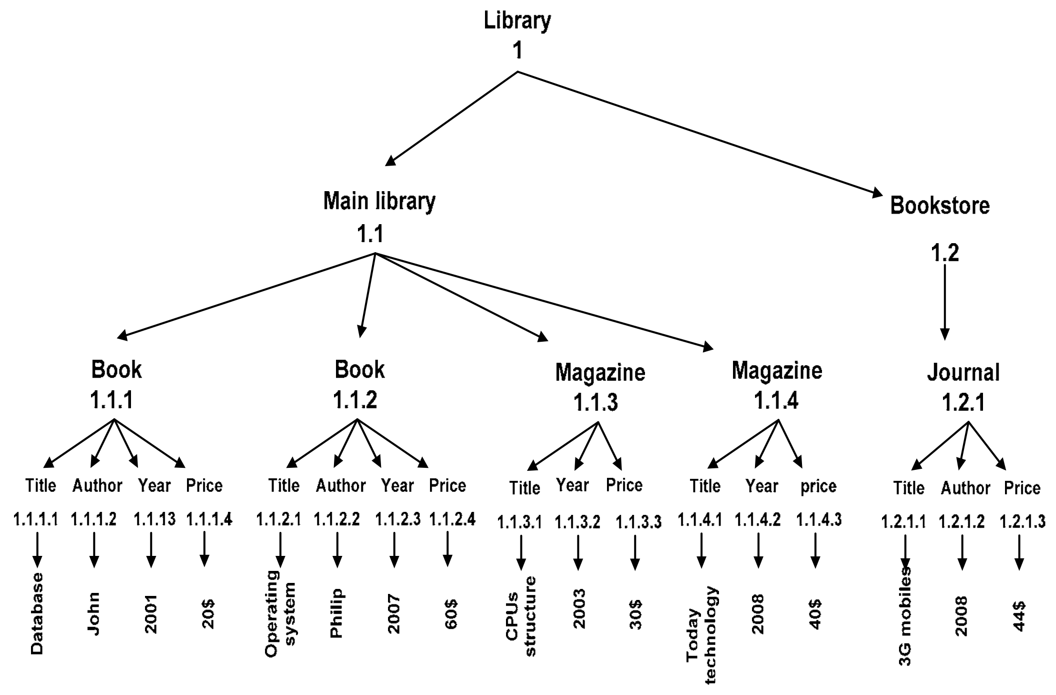


Figure 3.2: a tree representation for XML document in Figure 3.1

MXF parser traverses the above XML tree and labels all its leafs. It extracts all distinct elements: *Library, Main library, Bookstore, Book, Magazine, journal, title, author, year and price*. Then it generates a distinct binary number (tag) for each and creates a tag table; this table has two columns: tag name and tag binary. Tag name column stores all distinct elements in the document and tag binary column stores a unique binary number for each element as shown follows:

Tag name	**Tag**
Library	0000
Main library	0001
Bookstore	0010
Book	0011
Magazine	0100
Journal	0101
Title	0110
Author	0111
Year	1000
Price	1001

Table 3.1: the tag table for the XML documet in Figure 1.1

The idea behind creating binary tags is to make the splitting operation easy when the data block becomes full. We will see how splitting operation done in detail later in this chapter.

The second function of MXF parser is generating the Path Binary Label (BPL) and Path DEWEY (PD) for all paths from the root to each leaf node. MXF parser takes all paths one at a time, creates its BPL and PD. Then it passes these paths to the indexer one at a time. The following table shows all BPLs and PD for all paths in the XML tree above. Actually, we don't have this table in our MXF structure, but it is here to show the output of the MXF parser row by row.

path	path name	Path binary	Path DEWEY
1	*Main library /book/title*	0001/0011/01	1.1.1.1
2	*Main library /book/author*	0001/0011/01	1.1.1.2
3	*Main library /book/year*	0001/0011/1000	1.1.1.3
4	*Main library /book/price*	0001/0011/1001	1.1.1.4
5	*Main library /book/title*	0001/0011/01	1.1.2.1
6	*Main library /book/author*	0001/0011/01	1.1.2.2
7	*Main library /book/year*	0001/0011/1000	1.1.2.3
8	*Main library /book/price*	0001/0011/1001	1.1.2.4
9	*Main library /magazine/title*	0001/0100/0110	1.1.3.1
10	*Main library / magazine /year*	0001/0100/1000	1.1.3.2
11	*Main library / magazine /price*	0001/0100/1001	1.1.3.3
12	*Main library /magazine/title*	0001/0100/0110	1.1.4.1
13	*Main library / magazine /year*	0001/0100/1000	1.1.4.2
14	*Main library / magazine /price*	0001/0100/1001	1.1.4.3
15	*Book store*/ journal /*title*	0010/0101/0110	1.2.1.1
16	*Book store*/ journal /*year*	0010/0101/1000	1.2.1.2
17	*Book store*/ journal /*price*	0010/0101/1001	1.2.1.3

Table 3.2: the paths, their corresponding BPLs and DEWEY

3.2.2. MXF Indexer

This part consists of two main parts: MXF directory storage and MXF data storage.

3.2.2.1. MXF directory storage:

While storing XML data, MXF indexer creates directory blocks which contain indexes for data blocks. The idea of storing data using MXF is to store similar or almost similar paths in minimum number of data blocks; so retrieving similar data will cost less. In the next section we will explain in details how XML data is organized in MXF.

MXF directory is created in a multidimensional way for all generated binary paths. We will give an example to illustrate how multidimensional directory created later in this chapter.

3.2.2.2. MXF data storage:

This part of MXF is a set of tables that contain the raw XML document data. Any XML path is set of elements, relations, attributes and values. From this point of view, MXF is designed to be a set of tables namely:

- Tag table: to save the distinct elements with their distinct binary tags
- DEWEY table: to save the positional relations between the elements
- Attribute table: contains all attributes found in the XML document and their corresponding DEWEY, and
- Value table: contains the values of the all XML paths.

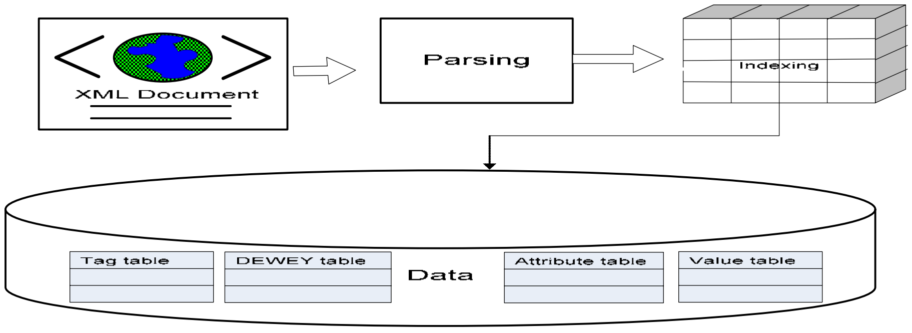


Figure 3.3: the MXF structure

MXF data storage is a set of tables each to save specific data that is generated from the MXF parser. BPLs generated from MXF parser are stored in data blocks and each block contains similar or almost similar BPL. The following algorithm will explain how MXF store BPLs

ALGORITHM 3.1: Store BPLs

INPUT: An empty data block, an empty directory block and BPLs to store

OUTPUT: BPLs stored in a MXF

```
1.     Read the first binary path
2.     Save it in the first data block
3.     Create its index and save it in a multidimensional space
4.     While NOT end of paths
5.       Read the next path
6.       Create the index of the path
7.       If the number of data blocks = 1 then
8.          Look for this index in the whole multidimensional space
9.          If found
10.            If there is enough space in the first data block then
11.               Save the path in the first data block
12.               Else
13.               Split the first data block
14.            End if
15.            Else
16.            Save its index in the multidimensional space
17.            If there is enough space in the first data block then
18.               Save the path in the first data block in
19.               Else
20.               Split the first data block
21.               End if
22.            Else
23.            Look for this index in the whole multidimensional space
24.            If found
25.               If the there is enough space in the data block to which the
                  index point then
26.               Save the path in that disk block
27.               Else
28.               Split this data block
29.               End if
30.            Else
31.            Create a new index
32.            Save the path in new data block
33.            Save the new index in the multidimensional space
34.            End if
35.         End if
36.      End if
37.   Loop
```

To explain how above algorithms works, let us take the following example

Example 4.3: suppose we have BPLs in Table 3.2. And for simplicity, we also suppose the capacity of the block is three BPLs (in real we measure the size of the block in KB). At the beginning we have one empty data block. The first three BPLs should be stored in this data block as in the following figure.

0001/0011/0110

0001/0011/0111

0001/0011/1000

Figure 3.4: MXF after saving the 1st BPL

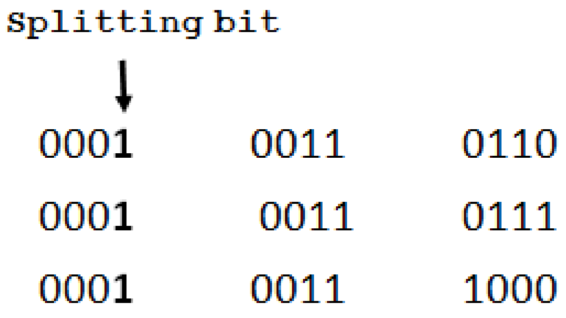
The fourth BPL 0001/0011/1001 cannot fit in the first data block because it is full; so we need to split these BPLs in to two data blocks.

To split the above data block, we have three paths in that block each path has three elements and each element is represented using three bits. If we look to these three paths we will see that they have two elements in common (the first element 0001 and the second element 0011), but they differ in the third element.

This means that the first two elements should appear in both indexes of the two new data blocks, but each if these indexes should have its own element in the third dimension.

So we start from the right most bit of the first element of the first path compare it with corresponding bits in the remaining paths. If there is a difference, split paths with zero's bits in a data block and paths with one's bits in another data block. If we reach the left most bits of that element without splitting, we move to the right most bit of the second element of the first path and repeat the same comparing with corresponding bits in the remaining paths. So, the above data block will be split as follows:

In the first step we compare the right most bits of the left most element (written in bold) in all paths, they are the same so we need to move to the previous bit.



000**1**	0011	0110
000**1**	0011	0111
000**1**	0011	1000

Figure 3.5: comparing the right most bits of the 1st element of the BPLs

Moving to the previous bit, they are also the same so we keep moving to the previous bit.

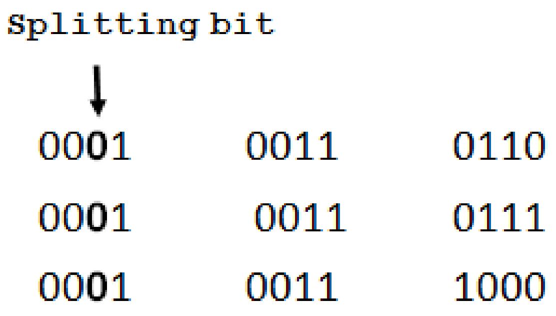


Figure 3.6: comparing the 2nd bits of the 1st element of the BPLs

We repeat the same comparisons and moving till we reach to different bits.

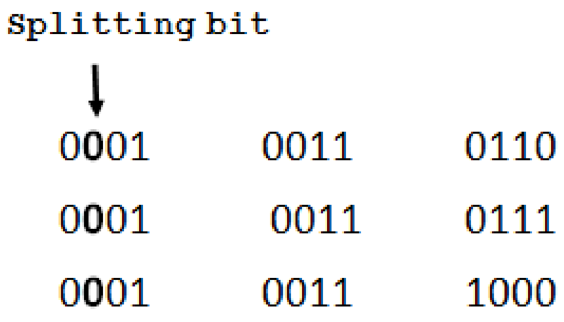


Figure 3.7: comparing the 3rd bits of the 1st element of the BPLs

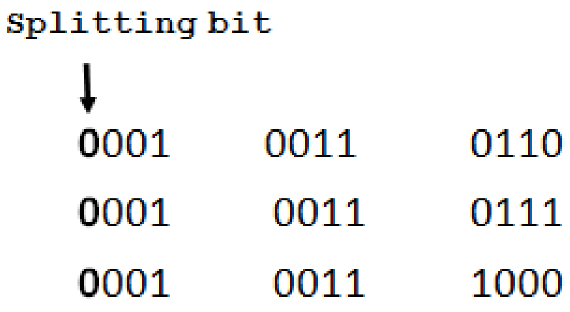


Figure 3.8: comparing the 4th bits of the 1st element of the BPLs

At this step, we reach the first bit of the first element without any splitting, so we move to the right most bit of the second element and repeat the same comparisons till we get difference.

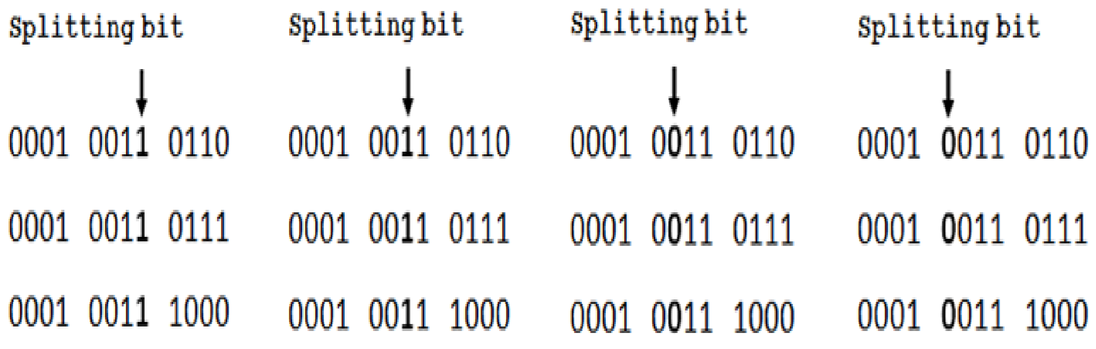


Figure 3.9: comparing the four bits of the 2nd element of the BPLs

Also we reach the left most bit of the second level without splitting, so we move to the right most bit of the third level as shown bellow

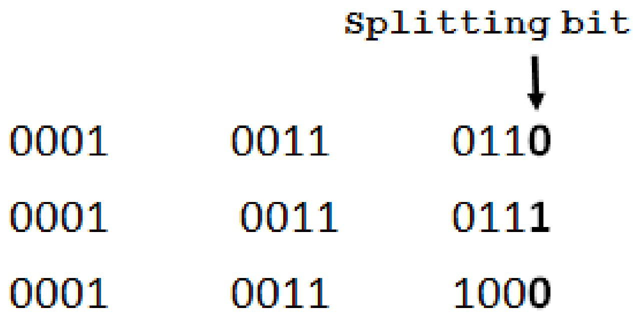


Figure 3.10: comparing the 1st bits of the 3rd element of the BPLs

In this step, the right most bits of the third element in the three paths are not the same; so according to this bit, the data block has to be split in to two new data block. The first block will contain the paths whose splitting bit is zero and the second block will contain the paths whose splitting bit is one as follows.

First block	Second block
0001 0011 011**1**	0001 0011 011**0**
	0001 0011 100**0**

Figure 3.11: splitting the previous block into two blocks

So, the index of the first data block will be **“0001 0011 ???1”** and the index of the second data block will be “**0001 0011 ???0**” and the BPL “0001 0011 1000” will be stored in the second block.

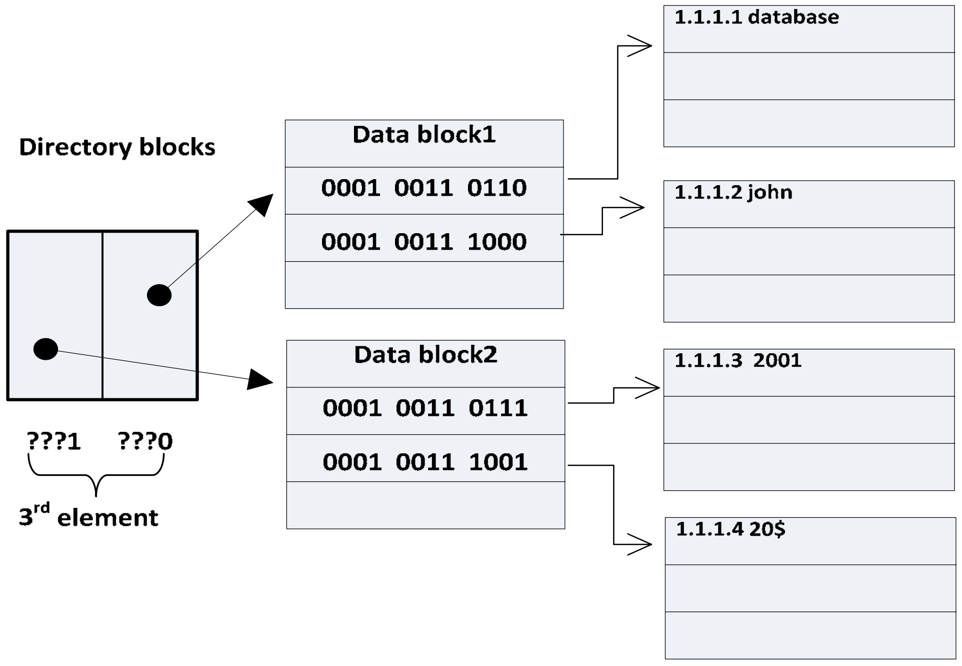


Figure 3.12: MXF after saving the 4th BPL

To save the next BPL “0001 0011 0110”, we see that our directory is split into two parts each has its index. So we have to compare this BPL with the two indexes to see where it will fit. By comparing this BPL with first index **“0001 0011 ???1”,** it is clear that this index is not the correct index for this BPL since they differ in the rightmost bit of the third element. By comparing this BPL with second index **“0001 0011 ???0”,** it is clear that this index is the correct index for this BPL since they share all elements of the path. After saving this BPL the our MXF will be as in the following figure.

Note: if there is a BPL in a data block similar to the BPL we need to save (as in this case), we will not store the new BPL again. Instead, we only save its value and DEWEY in the value blocks without saving the BPL again. This operation eliminate BPLs repeat. As we dealing with this BPL.

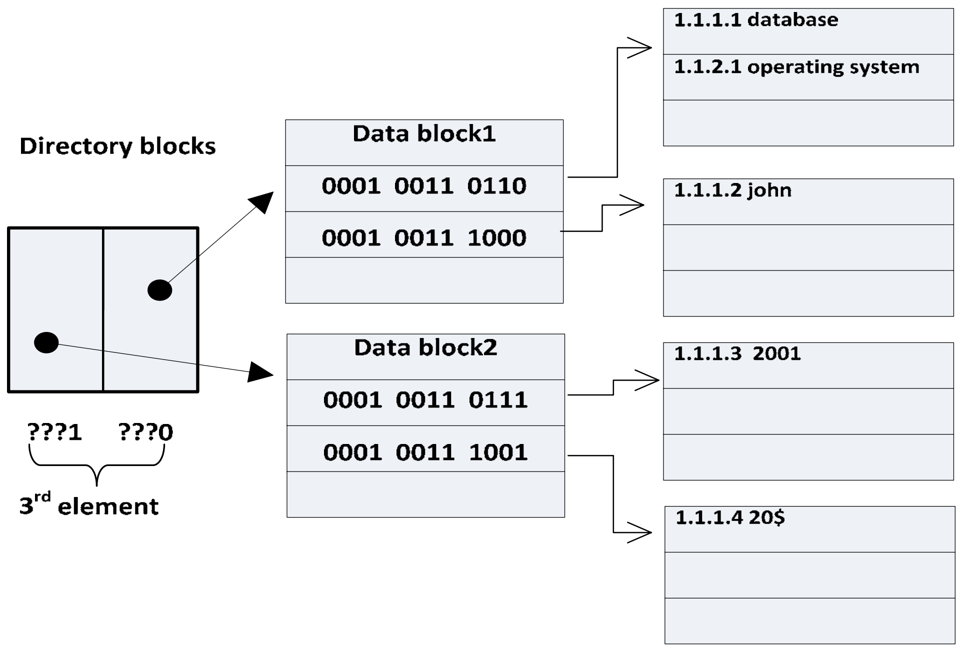


Figure 3.13: MXF after saving the 5th BPL

The same scenario should be done with the next BPL “0001/ 0011/0111”as shown in a figure bellow

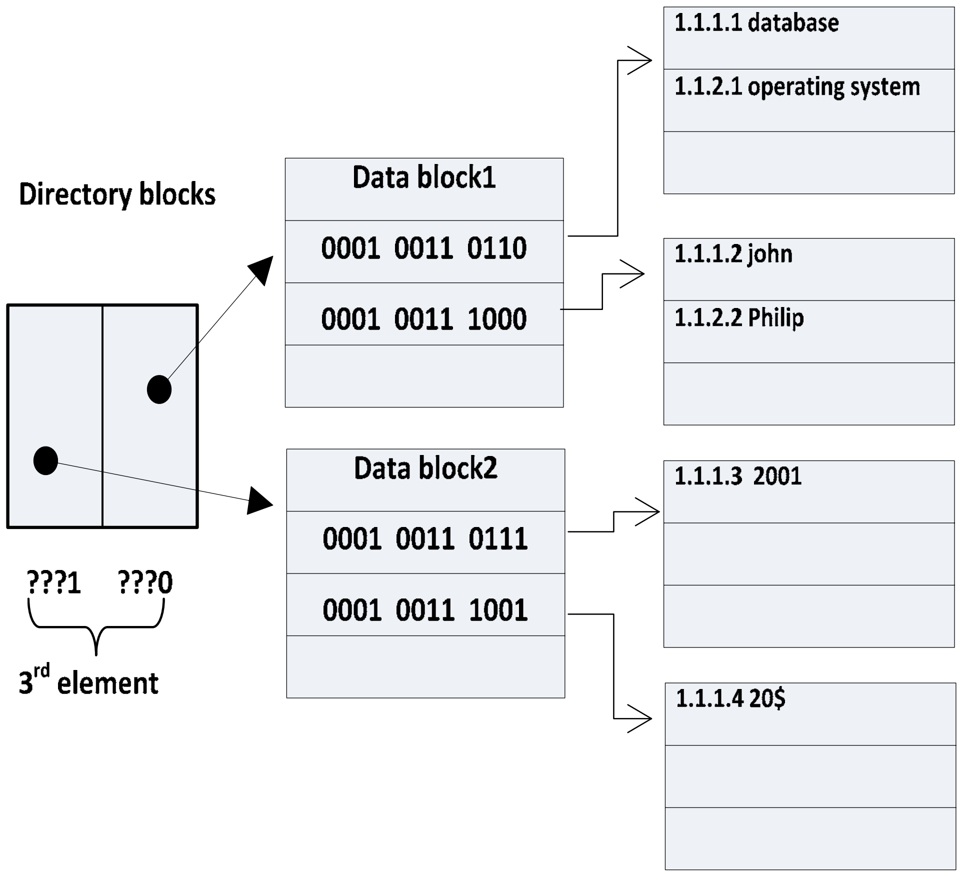


Figure 3.14: MXF after saving the 6th BPL

The same scenario should be done with the next two BPLs “0001/ 0011/1000” and “0001 0011 1001”as shown in a figure bellow.

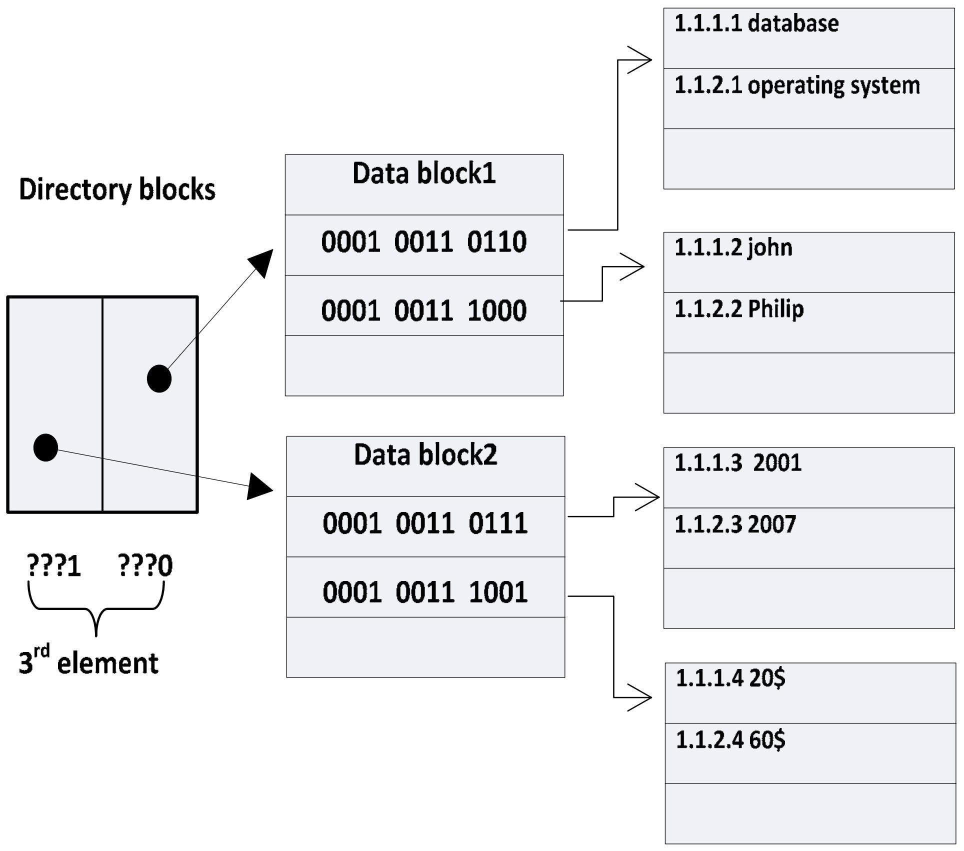


Figure 3.15: MXF after saving the 7th and 8th BPLs

Till this point we can see that all BPLs stored share the first and the second elements, but they differ in the third element. Now, to store the next BPL “0001 0100 0110” we see that its second element is different from all previous BPLs so it has not a previous index. To create a new index for it we have to do a split in the directory as the following figure shows.

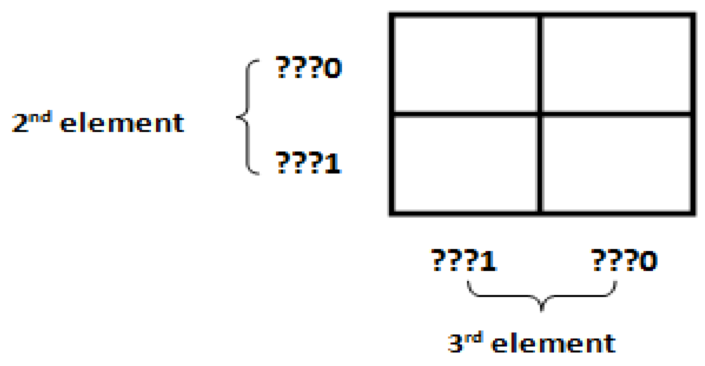


Figure 3.16: MXF directory after splitting the second dimension

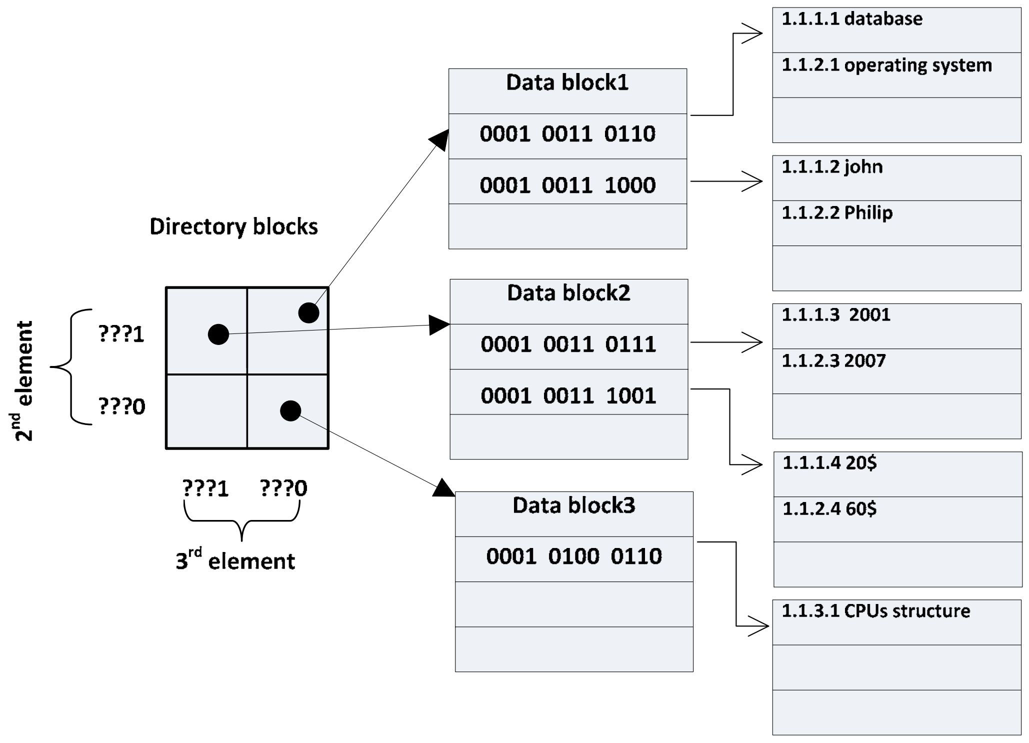


Figure 3.17: MXF after saving the 9th BPL

To save the next BPL "0001 0100 1000", we can see that its third element start with "0" and its second element start with "0", so it can fit in the directory above as shown in the following figure.

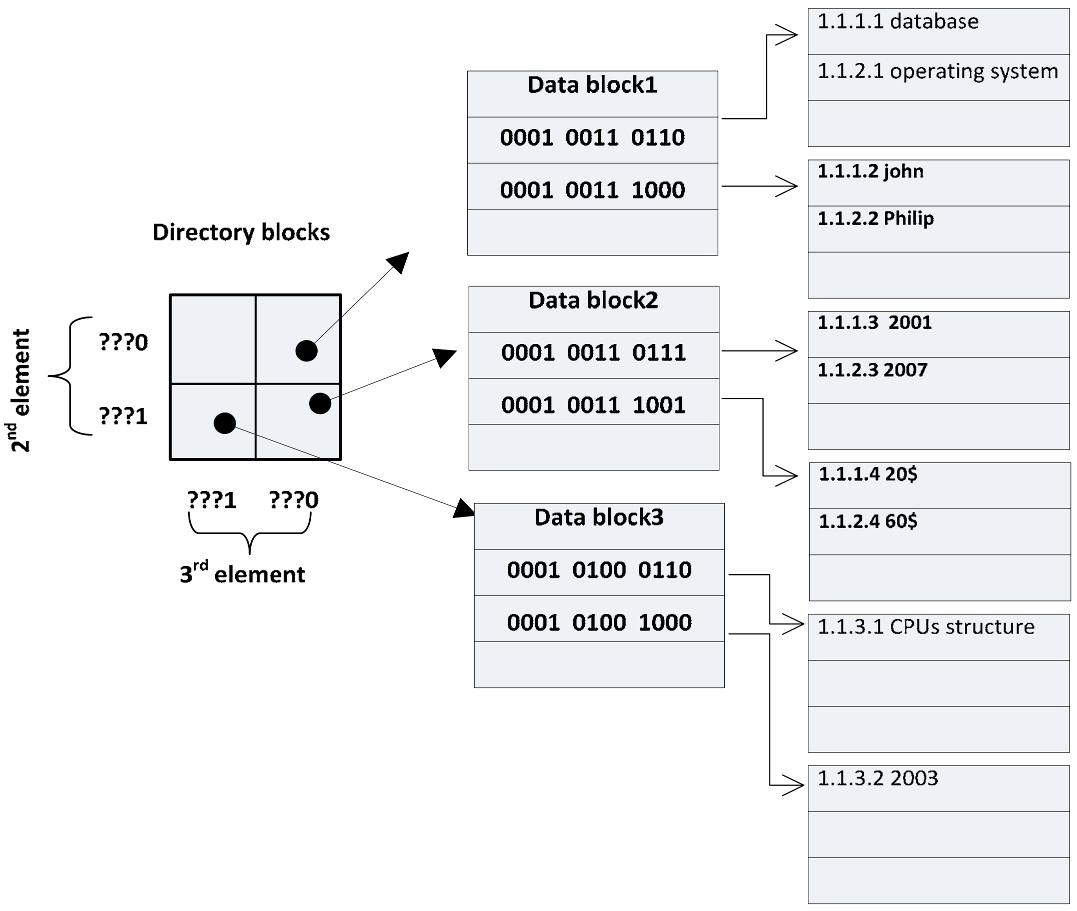


Figure 3.18: MXF after saving the 10th BPL

To save the next BPL "0001 0100 1001", we can see that its third element start with "1" and its second element start with "0", so it can fit in the directory above as shown in the following figure.

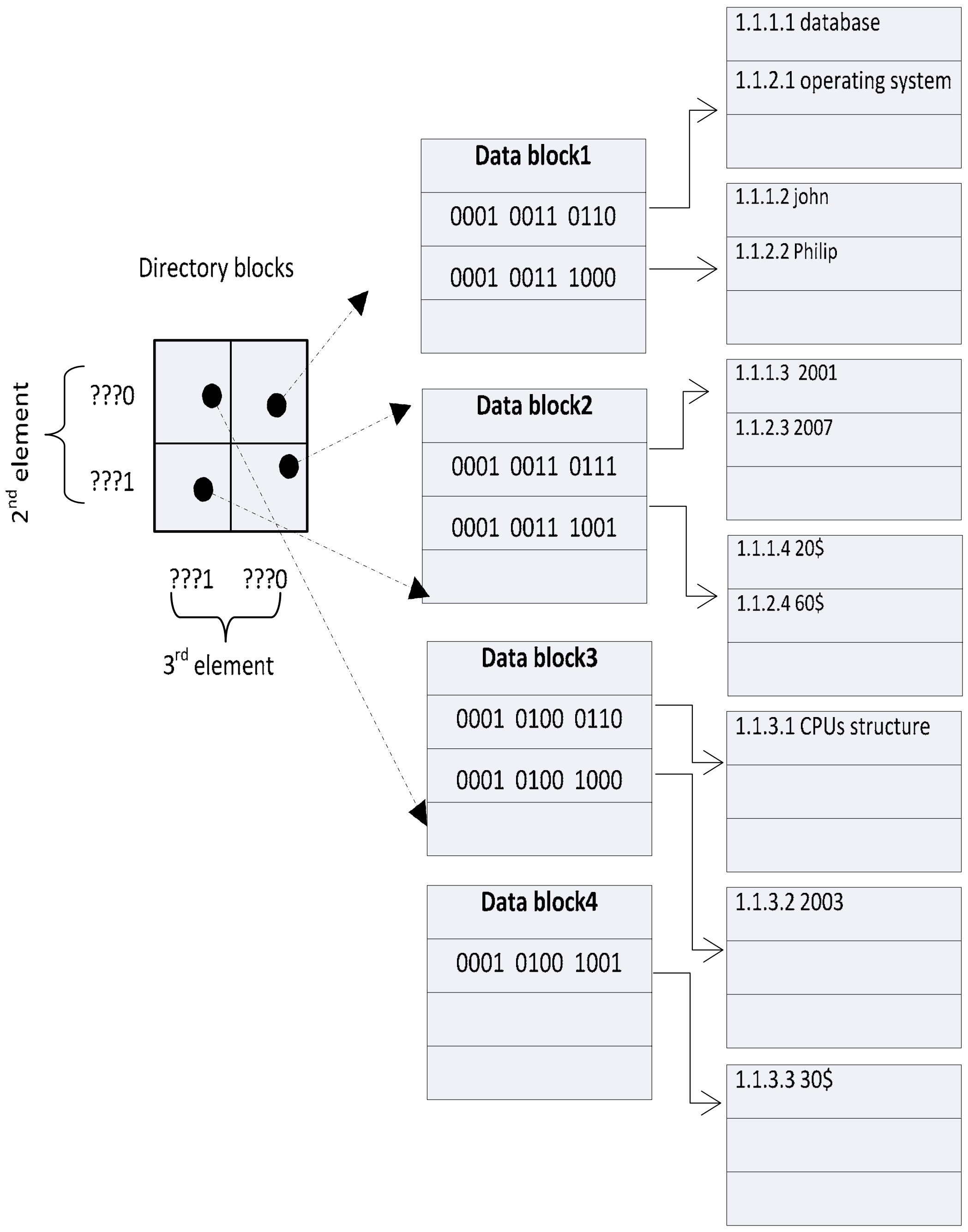


Figure 3.19: MXF after saving the 11th BPL

To save the next three BPLs “0001 0100 0110”, “0001 0100 1000”, and “0001 0100 1001”, we can see that it can fit in data block3 and data block4 as shown in the following figure

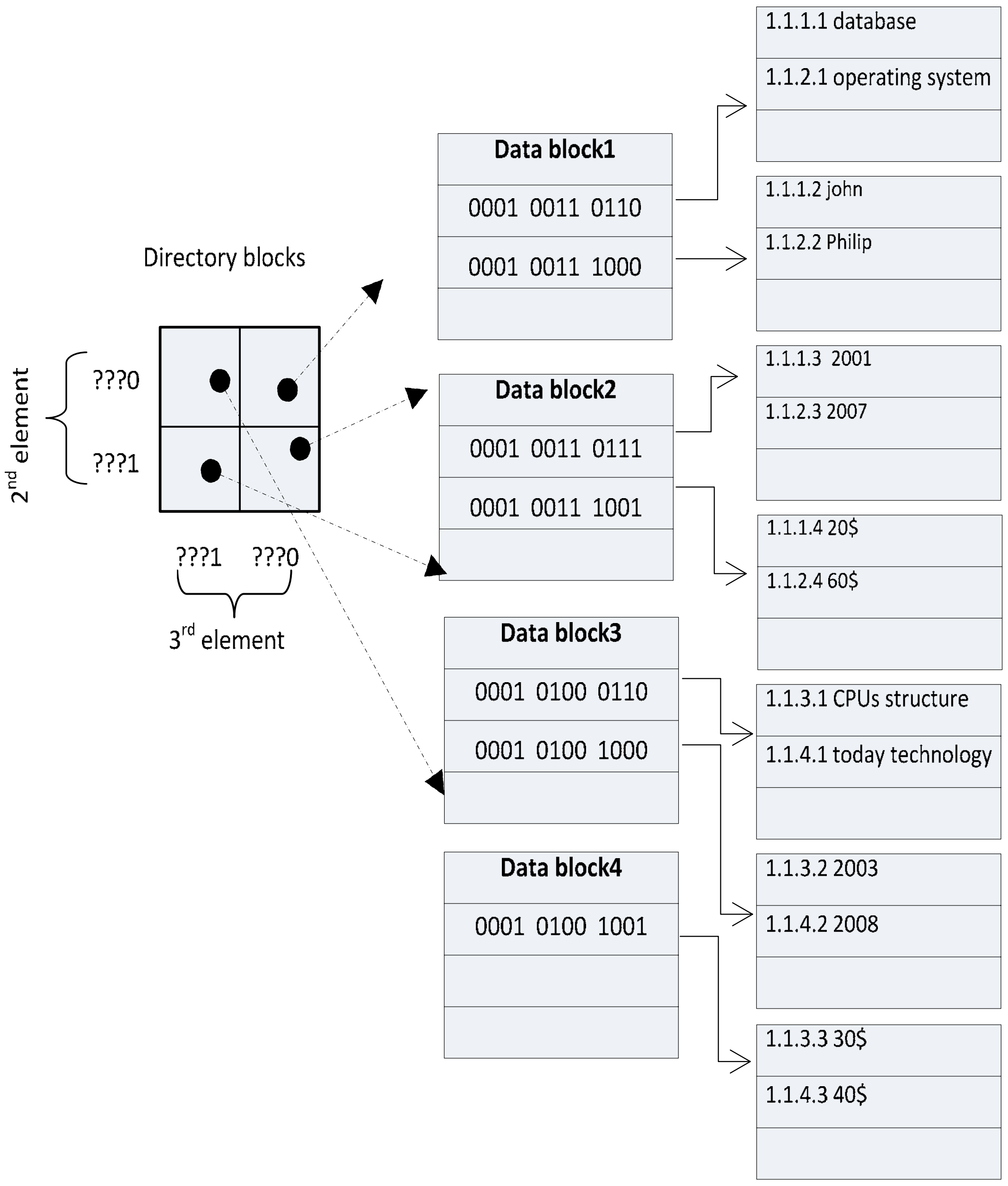


Figure 3.20: MXF after saving the 12th, 13th and 14th BPLs

To store the next BPL”0010 0101 0110”, we can see that this BPL differ from the all previous BPLs in the first element, so we have to create a new index for this BPL. The 1st element we want to add to the above directory will create the 3rd dimension of the directory. Note that before this point the directory we have is two dimensions directory. The figure bellow shows the directory after creating the 3rd dimension.

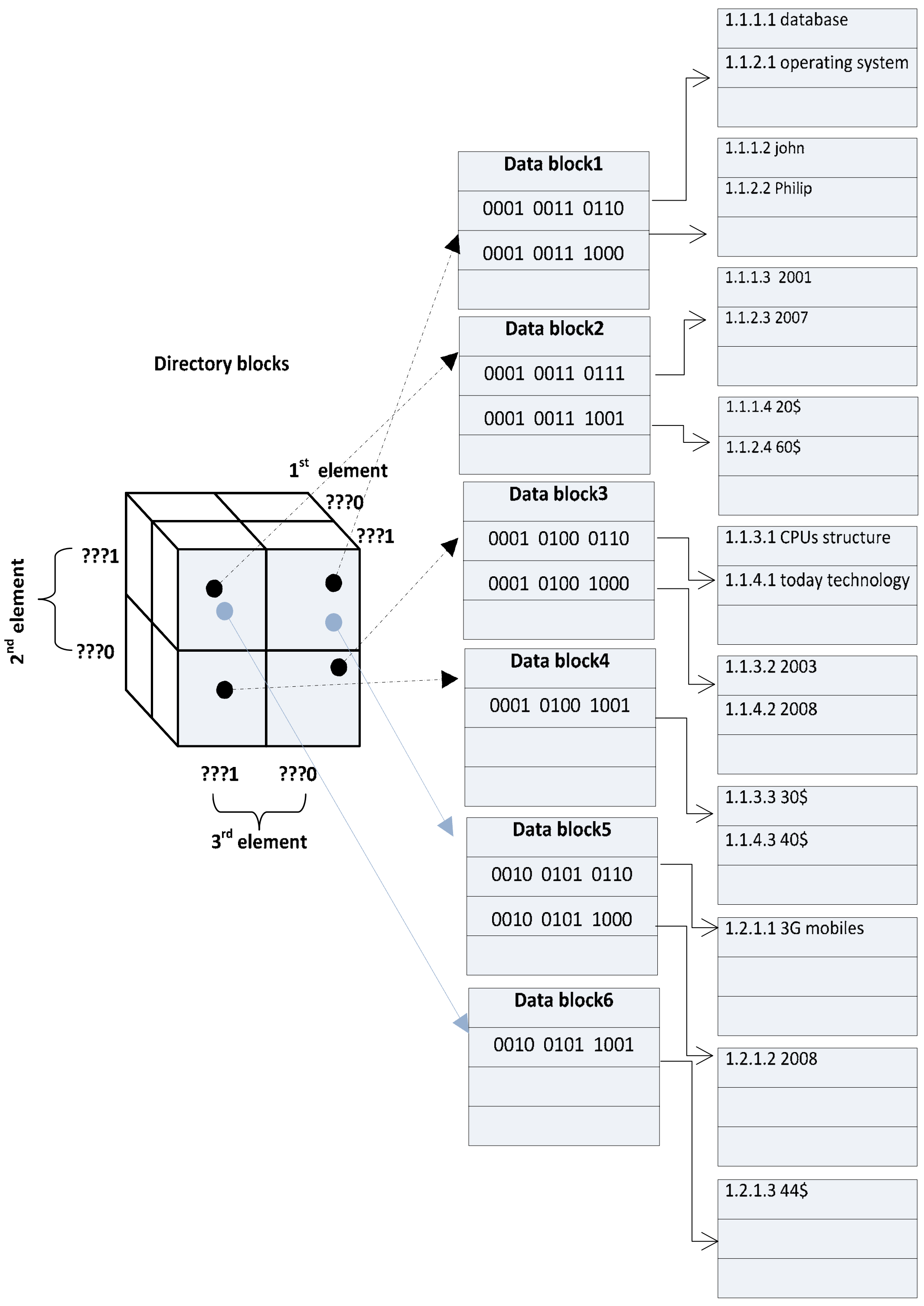


Figure 3.21: MXF after saving the remaining BPLs with a three dimensions directory

3.3. Main operation on MXF

The main operations that can be applied on the MXF are: searching, insertion and deletion.

3.3.1. Searching

The searching for BPL in the MXF can be done in two main steps:

- The first step is the search in the MXF directory (searching in the XMF index space): given a BPL we decompose it into many levels equals to the depth of the BPL (are number of nodes in the BPL). Then we start from the right most level looking for its corresponding dimension in the directory space. The number of searching steps (from the right most to the left) equals to the number of the dimensions which also equals to the number of levels in the document. As a result of this step, we will get the targeted directory block (the index) if the BPL matches an index in the directory blocks.
- The second step: the directory block gained in the first step will point to the corresponding data block.

Algorithm 3.2 explains the searching operation.

Given MXF directory_blocks organized in multidimensional space with n dimensions (where n is the largest level in the XML document and each dimension is composed on m parts), also given MXF data_blocks (the blocks where the data is stored) and BPL (the path we want to looking for in the MXF).

Let *BPL_level$_i$* denotes to the ith level in the BPL

Let dimension_part_$_j$ denotes to the j part in the dimension i

Let path_address denotes to the address of the path in the MXF data_blocks

Algorithm 3.2 BPL SEARCH

```
Input: directory blocks, data blocks and BPL
Output:  BPL address if BPL is found else returns -1
      1.    BPL_address=" "   // BPL address in the data blocks
      2.    For each BPL_level_i in the BPL  do
      3.        Found =false
      4.        For each dimension_part_ j in the dimension  i do
      5.           If dimension_part_ j = BPL_level_i or dimension_part_ j
                   contains BPL_level_i then
      6.              BPL_address = BPL_address & dimension_part_ j
      7.              Found= true
      8.           End if
      9.           If found = true then exit for
      10.       End for
      11.       If found = false then exit for
      12.       End for
      13.       If found =true then
      14.       For i= BPL_address to BPL_address + block_size do
      15.       If BPL= path_address(i) then
      16.       return I
      17.   Exit for
      18.   return -1
```

3.3.2. Insertion

To insert a BPL, two main steps have to be done:

- The first step is searching for a suitable directory block where the index of this BPL will be found (if the index already exists) or the index will be generated and stored (if it does not already exist). We use the procedure we used in searching algorithm.

- The second step is to go to the data block (whose index is in the directory block we found in the first step) and store this BPL if there is enough space or split that data block in to two data blocks if it is full.

Algorithm 3.3 explains the insertion operation.

Given the same MXF structure explained in the search algorithm, and BPL (the path we want to insert into the MXF).

Algorithm 3.3 BPL INSERTION

```
Input: directory blocks, data blocks and BPL to be inserted
Output:  MXF with a BPL inserted in
 19.     BPL_address=" "   // BPL address in the data blocks
 20.     For each $BPL\_level_i$ in the BPL  do
 21.       Found =false
 22.       For each part dimension_part_ $_j$ in the dimension  i do
 23.         If dimension_part_ $_j$ = $BPL\_level_i$ or dimension_part_ $_j$ contains
              $BPL\_level_i$ then
 24.          BPL_address = BPL_address & dimension_part_ $_j$
 25.          Found= true
 26.          Exit for
 27.          End if
 28.       If found = false then
 29.          make a new part in the dimension I  name it BPL_level $_i$
 30.          BPL_address = BPL_address & $BPL\_level_i$
 31.          End if
 32.       End for
 33.     End for
 34.     If found = true then
 35.       If the data block (BPL_Address) is not full then
 36.       Insert BPL in that data_block
 37.       Else
 38.       Split data block (BPL_Address)
 39.       Update the directory
 40.       End if
 41.     else
 42.       create a new data_block  give its first path the address
           data_block_numbers * block_ size
 43.       path (data_block_numbers * block_ size)=BPL // insertion BPL into
           the new data_block
 44.       Update the directory
 45.     End if
```

3.3.3. Deletion

To delete a BPL, two main steps have to be done:

- The first step is searching for a suitable directory block where the index of this BPL will be found. We use the procedure we used in searching algorithm.
- The second step is to go to the data block (whose index is in the directory block we found in the first step) and delete this BPL.

Algorithm 3.4 explains the deletion operation.

Given the same MXF structure explained in the previous algorithm, and BPL (the path we want to delete from the MXF).

ALGORITHM 3.4 BPL INSERTION

```
Input: directory blocks, data blocks and BPL to be deleted
Output:  MXF with a BPL deleted from it in
   1.   BPL_address=" "   // BPL address in the data blocks
   2.   For each $BPL\_level_i$ in the BPL  do
   3.      Found =false
   4.      For each part dimension_part_ $_j$ in the dimension  i do
   5.       If dimension_part_ $_j$ = $BPL\_level_i$ or dimension_part_ $_j$ contains
              $BPL\_level_i$ then
   6.            BPL_address = BPL_address & dimension_part_ $_j$
   7.            Found= true
   8.            Exit for
   9.            End if
   10.     End for
   11.  End for
   12.  If found = true then
   13.     For i= BPL_address to BPL_address + block_size do
   14.     If path (i)=BPL  then path (i)= "  " // deletion the BPL from the
           data_block
   15.     End if
   16.  End if
```

CHAPTER FOUR

EXPERIMENTAL RESULTS AND ANALYSIS

To study the performance of MXF a number of parameters have to be taken into account. The most important parameters are: the numbers of disk blocks that are used to save the data, the density of the file i.e. how disk blocks dense are, and the query response time.

Since we will use these terms (data-depth, data-width, data-shape and data-variety) in this chapter frequently, let us give brief definitions of each term.

- Data- depth: the number of levels in the BPL that is the number of nodes from the root to the leaf node.
- Data- width: the average number of nodes in a level in an XML document.
- Data- shape: is it width or depth data.
- Data -variety: the number of distinct BPLs in an XML document.

We experimented MXF with different datasets to study its performance. Sections 4.1, 4.2, 4.3 and 4.4 explain the environment, dataset, query set and performance metrics that are used respectively. Section 4.5 explains the experiments and their results. Section 4.6 summaries this chapter.

4.1. Environment

Experiments were performed on a core 2 due 2.2 MHZ processor with windows vista home. This workstation has 2 GB of memory and 160 GB of hard disk. MXF was designed using MS visual basic 2005.

4.2. Dataset

Nine datasets were chosen from real and synthetic data. These datasets are classified into two groups according to: data-variety and data-shape. Since the size of the data is an important factor, we use three groups (a GA, a GB and a GC) from each dataset where the GA has a smaller size than GB and GB has a smaller size than GC as shows in Table 4.1.

4.3. Query sets

Several simple path query sets will be applied to our datasets. These query sets will be mentioned later with corresponding datasets.

4.4. Performance metrics

As we mentioned before, we will use three performance metrics:

- **Number of disk blocks**: how many disk blocks that are used to save data, fewer number of disk blocks means better performance.
- **File density**: the capacity of the data within data blocks over the total capacity of these data blocks, and high disk blocks density is an indicator for better performance.
- **Query response time**: The number of disk blocks returned to answer a query.

Number	Dataset Name	Dataset version	Size in MB
1	SWISSPORT.xml	SWISSPORT_GC.xml	12.40
		SWISSPORT_GB.xml	6.89
		SWISSPORT_GA.xml	2.41
2	UWM.xml	UWM_GC.xml	9.49
		UWM_GB.xml	4.73
		UWM_GA.xml	2.28
3	PARTSUPP.xml	PARTSUPP_GC.xml	10.94
		PARTSUPP_GB.xml	4.37
		PARTSUPP_GA.xml	2.19
4	DBLP.xml	DBLP_GC.xml	10.76
		DBLP_GB.xml	5.74
		DBLP_GA.xml	1.36
5	WSU.xml	WSU_GC.xml	8.31
		WSU_GB.xml	3.94
		WSU_GA.xml	1.98
6	CUSTOMER.xml	CUSTOMER_GC.xml	10.07
		CUSTOMER_GB.xml	5.03
		CUSTOMER_GA.xml	1.51
7	PERSONS.xml	PERSONS_GC.xml	8.75
		PERSONS_GB.xml	4.37
		PERSONS_GA.xml	1.45
8	SIGMODRECORD.xml	SIGMODRECORD_GC.xml	13.05
		SIGMODRECORD_GB.xml	4.35
		SIGMODRECORD	1.45
9	ORDERS.xml	ORDERS_GC.xml	11.38
		ORDERS_GB.xml	5.25
		ORDERS_GA.xml	2.62

Table 4.1: the datasets used for our experiments

4.5. Experiments

For each dataset mentioned above, we assign a separate paragraph explaining the density of the file, the number of disk blocks used and the query response time.

4.5.1. Classification of data according to the data variety

We have implemented three dataset sets according to data variety (set A, set B and set C)

- **Set A**: the data sets used in this set are: three versions of SWISSPORT.xml (SWISSPORT_GC.xml, SWISSPORT_GB.xml and SWISSPORT_GA.xml). This set has the largest number of distinct BPLs among the three sets.
- **Set B**: the data sets used in this set are: three versions of UWM.xml (UWM_GC.xml, UWM_GB.xml and UWM_GA.xml)
- **Set C**: the data sets used in this set are: three versions of PARTSUPP.xml (PARTSUPP_GC.xml, PARTSUPP_GB.xml and PARTSUPP_GA.xml). This set has the smallest number of distinct BPLs among the three sets.

4.5.1.1. Set A datasets

Three groups will be implemented in this set:

4.5.1.1.1. Group C dataset

We have taken "**SWISSPORT_GC.xml**" database; its size is 12.40 MB with 79 distinct elements and 85 number of distinct paths. The number of disk blocks when the size of the block is 2 KB is 1298; the number of disk blocks when the size of the block is 4 KB is 690 and the number of disk blocks when the size of the block is 6 KB is 491 as shown in the following chart.

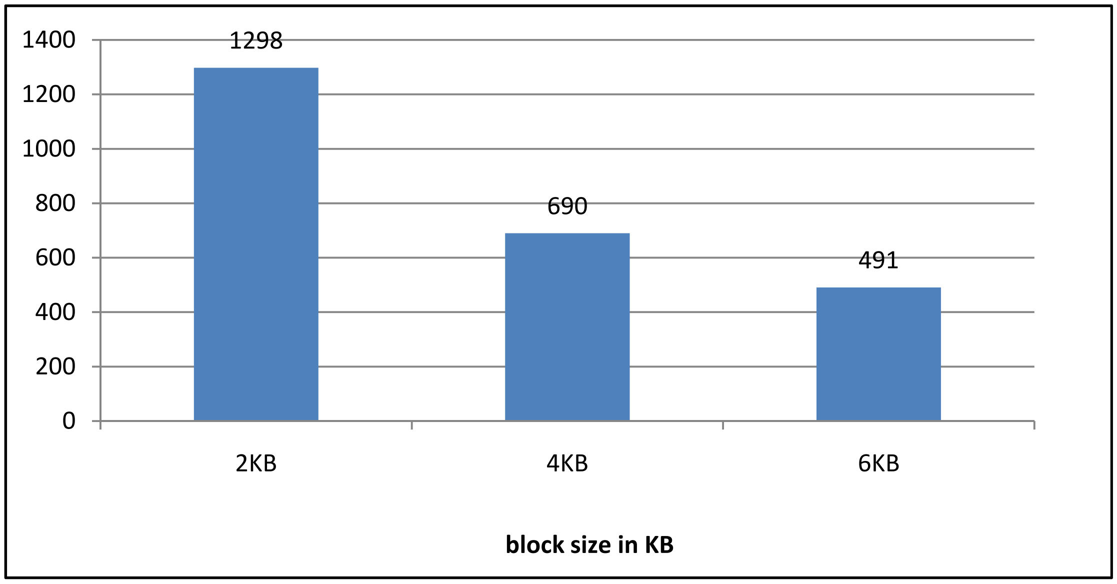


Figure 4.1: the number of blocks used to save SWISSPORT_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 93.88%, when the size of the block is 4 KB, the density of the file is 88.30 %; and the file density when the size of the block is 6KB is 82.72% as shown in the chart bellow.

We can see that as the size of the block becomes larger; its density becomes smaller because the data block needs more data to be store in it to be denser. But we have to note an important issue which belongs to the nature of the data itself. If the number of the distinct paths is very large here using larger data blocks means less density; but if the number of distinct paths is small; using larger data blocks size will not affect the density of the file. The second case is the usual case in XML document.

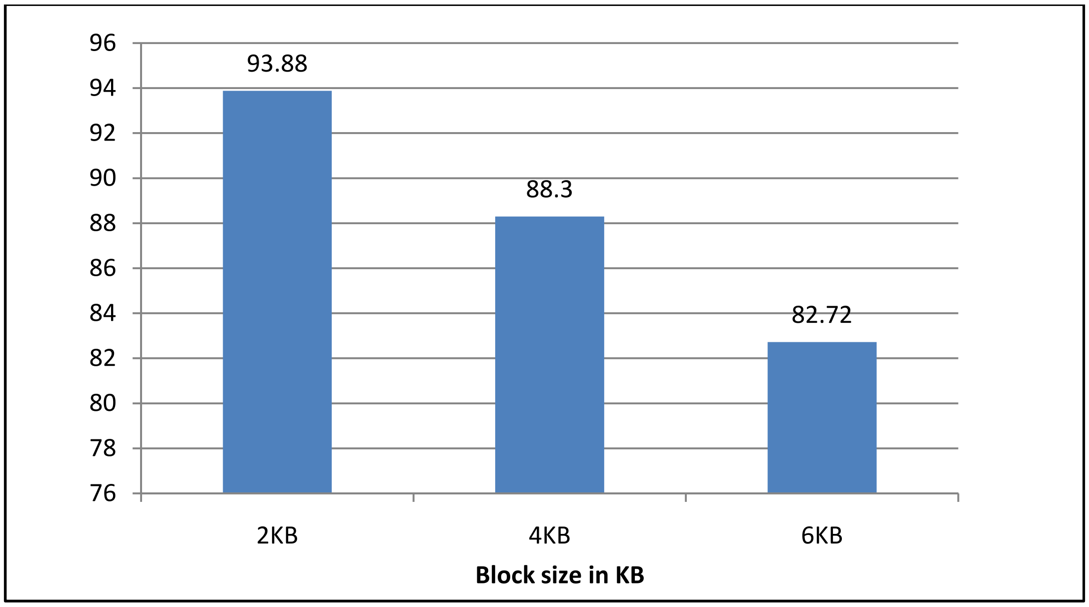


Figure 4.2: the file density when saving SWISSPORT_GC.xml dataset

4.5.1.1.2. Group B dataset

We have taken "**SWISSPORT_GB.xml**". Its size is 6.89 MB with 79 distinct elements and 85 distinct paths 85. The number of disk blocks when the size of the block is 2 KB is 817, the number of disk blocks when the size of the block is 4 KB is 447 and the number of disk blocks when the size of the block is 6 KB is 328 as shown in the following chart.

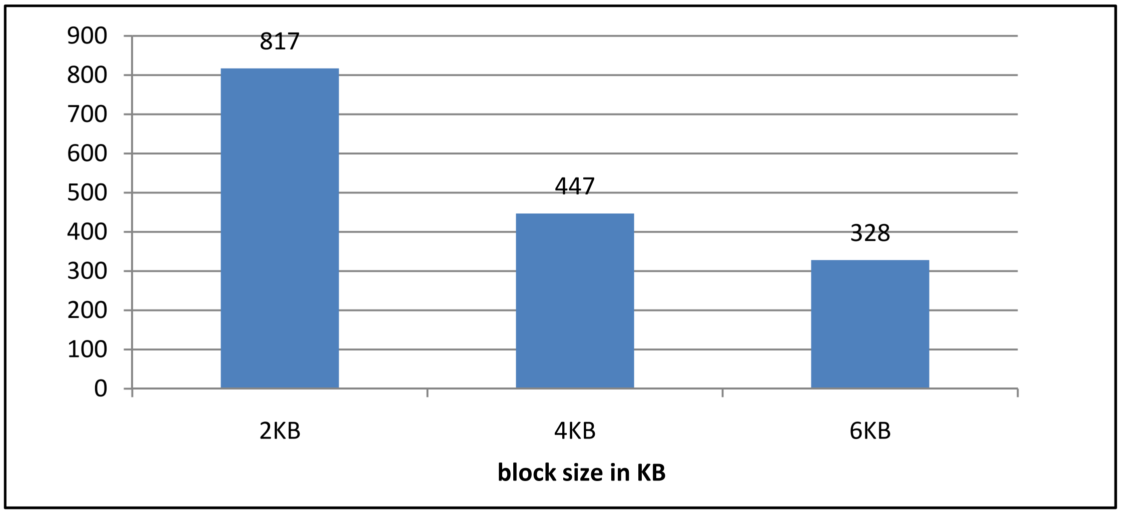


Figure 4.3: the number of blocks used to save SWISSPORT_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 89.29%, when the size of the block is 4 KB, the density of the file is 81.60 %; and the file density when the size of the block is 6KB is 74.14% as shown in the chart bellow

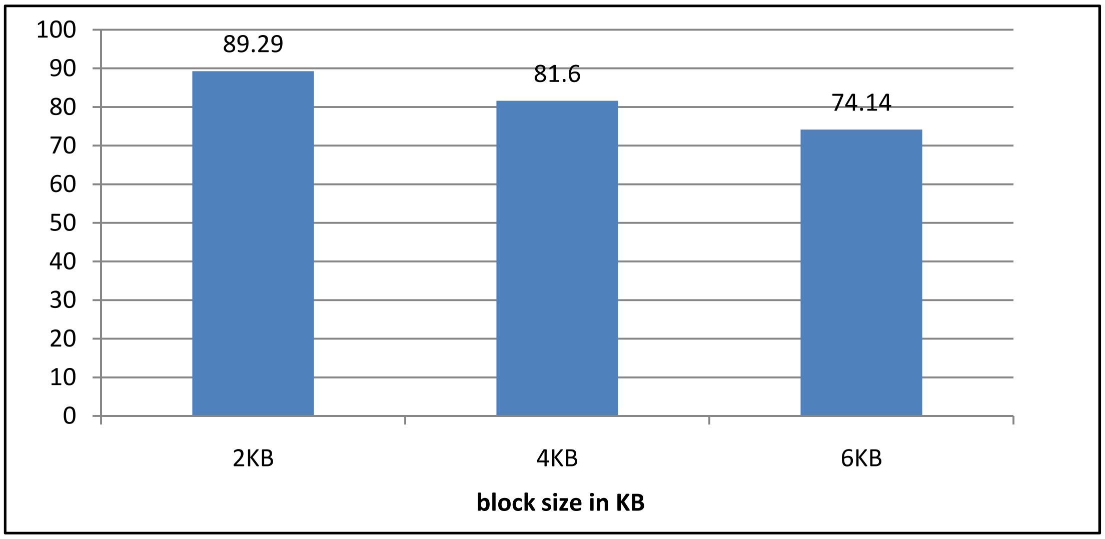


Figure 4.4: the file density when saving SWISSPORT_GB.xml dataset

4.5.1.1.3. Group A dataset

We have taken "**SWISSPORT_GA.xml**" database; its size is 2.41 MB with 79 distinct elements and 85 distinct paths. The number of disk blocks when the size of the block is 2 KB is 325, the number of disk blocks when the size of the block is 4 KB is 200 and the number of disk blocks when the size of the block is 6 KB is 163 as shown in the following chart

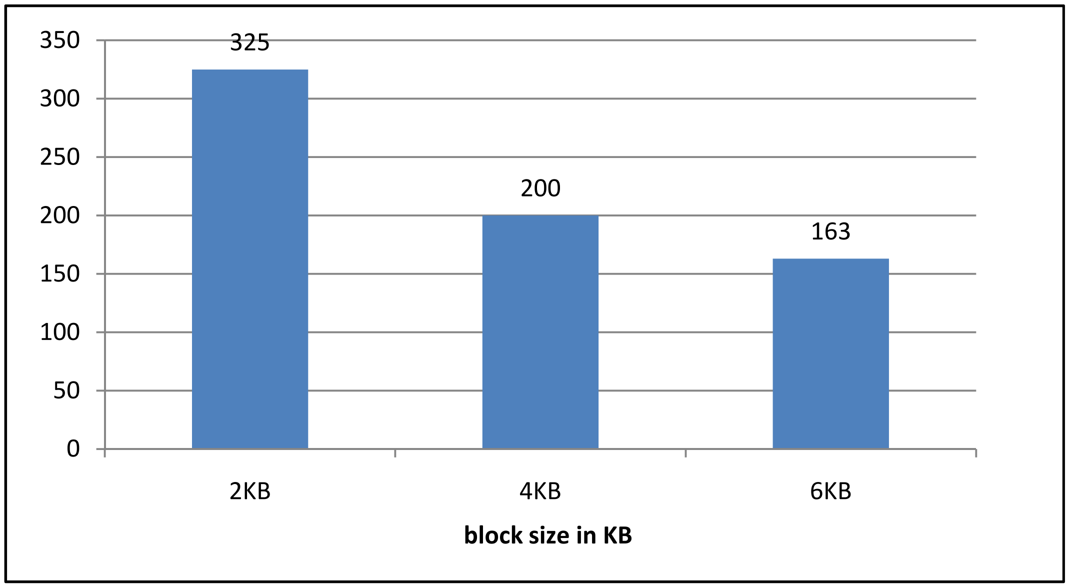


Figure 4.5: the number of blocks used to save SWISSPORT_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 89.29%, when the size of the block is 4KB, the density of the file is 81.60 %; and the file density when the size of the block is 6KB is 74.14% as shown in the chart bellow

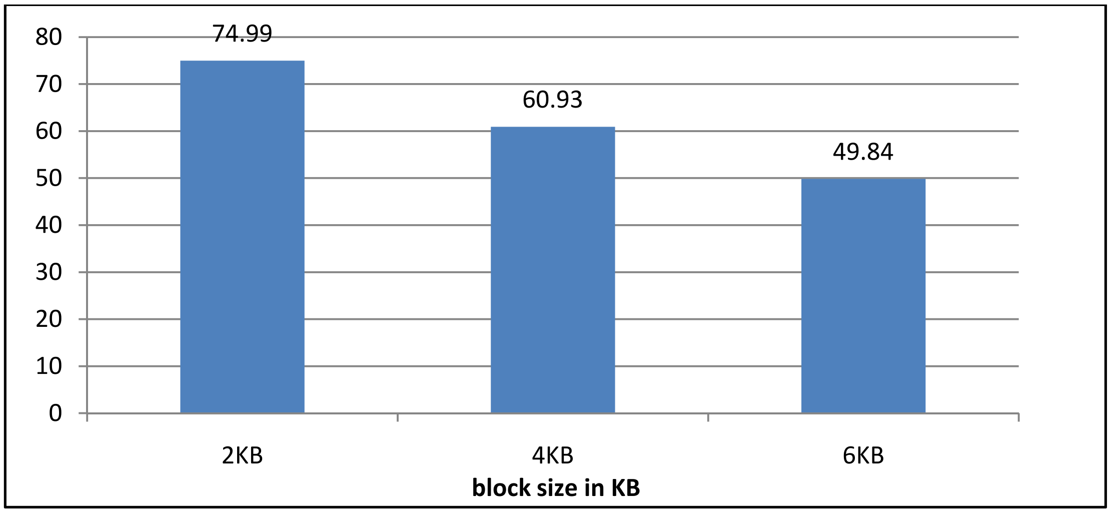


Figure 4.6: the file density when saving SWISSPORT_GA.xml dataset

It is shown from the above figures that the density of the file becomes less as the size of the data becomes smaller; and when the size of the data becomes larger the data blocks becomes denser.

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
SWISSPORT.xml	12.40	1298	690	491	93.88	88.30	82.72
	6.89	817	447	328	89.29	81.60	74.14
	2.41	325	200	163	74.99	60.93	49.84

Table 4.2: a summery table shows the density file and the number of blocks used when saving SWISSPORT.xml

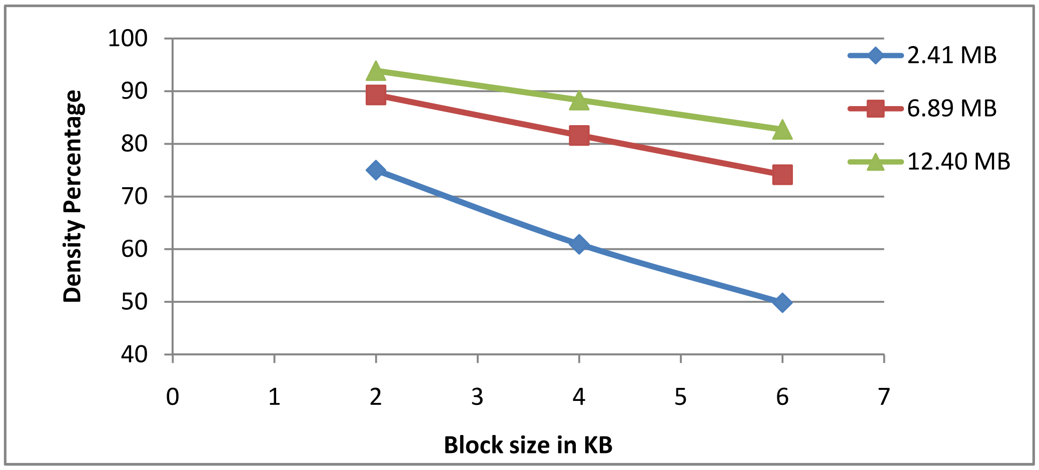


Figure 4.7: the density of the MXF when saving SWISSPORT.xml datasets

According to the response time of the queries applied on the three versions of "**SWISSPORT.xml**" database, we have applied these simple path queries. These queries are Entry/AC, Entry/Mod, Entry/Descr, Entry/Species, Entry/Org, Entry/Ref/Comment, Entry/Ref/DB and Entry/Ref/MedlineID. These queries will be given a number from 1 to 8 as they are ordered above as in the following table:

Query number	**Number of disk blocks used**		
	2kb	**4 KB**	**6 KB**
1	16	8	6
2	40	20	14
3	19	10	7
4	14	8	5
5	129	65	44
6	15	8	6
7	21	11	8
8	21	11	3

Table 4.3: shows the query response time for the query set mentioned above.

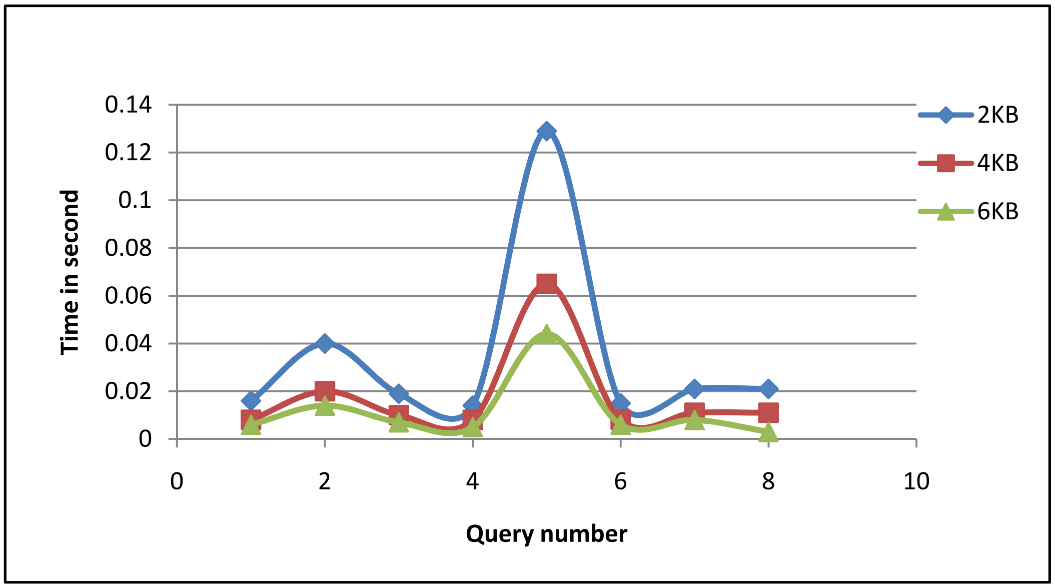


Figure 4.8: query responce time when queyring SWISSPORT.xml

The dominant factor of query response time is the number of disk blocks that are used to save the data

4.5.1.2. Set B datasets

Three groups will be implemented in this set:

4.5.1.2.1. Group C dataset

As a dataset with average distinct paths, we have chosen "**UWM_GC.xml**" database; its size is 9.46 MB with 21 distinct elements and 16 distinct paths. The number of disk blocks when the size of the block is 2 KB is 1129, the number of disk blocks when the size of the block is 4 KB is 575 and the number of disk blocks when the size of the block is 6 KB is 388 as shown in the following chart

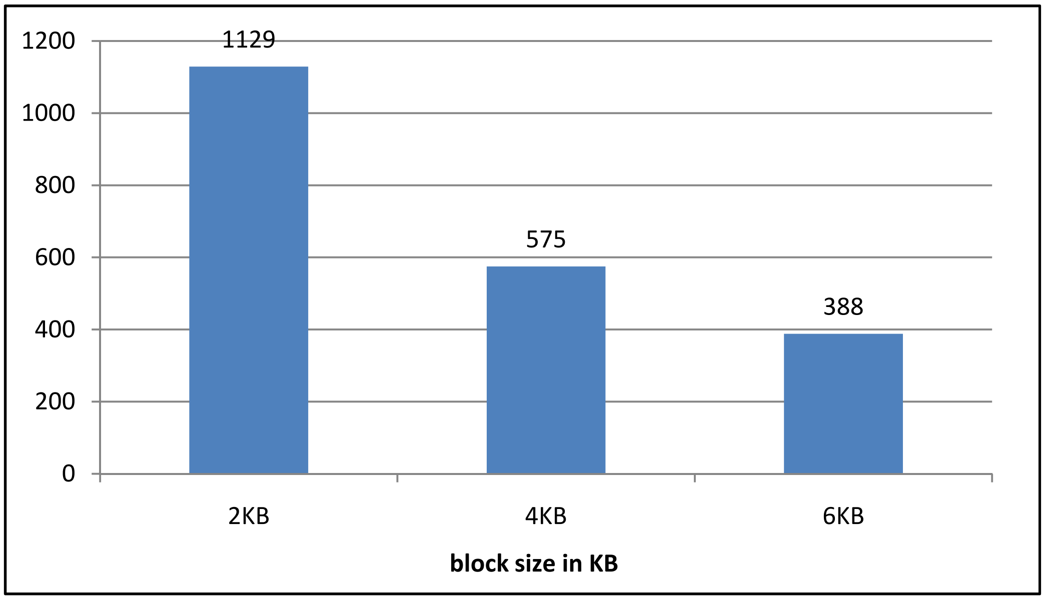


Figure 4.9: the number of blocks used to save UWM_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 98.78%, when the size of the block is the density of the file is 96.97 %; and the file density when the size of the block is 6KB is 95.81% as shown in the chart bellow

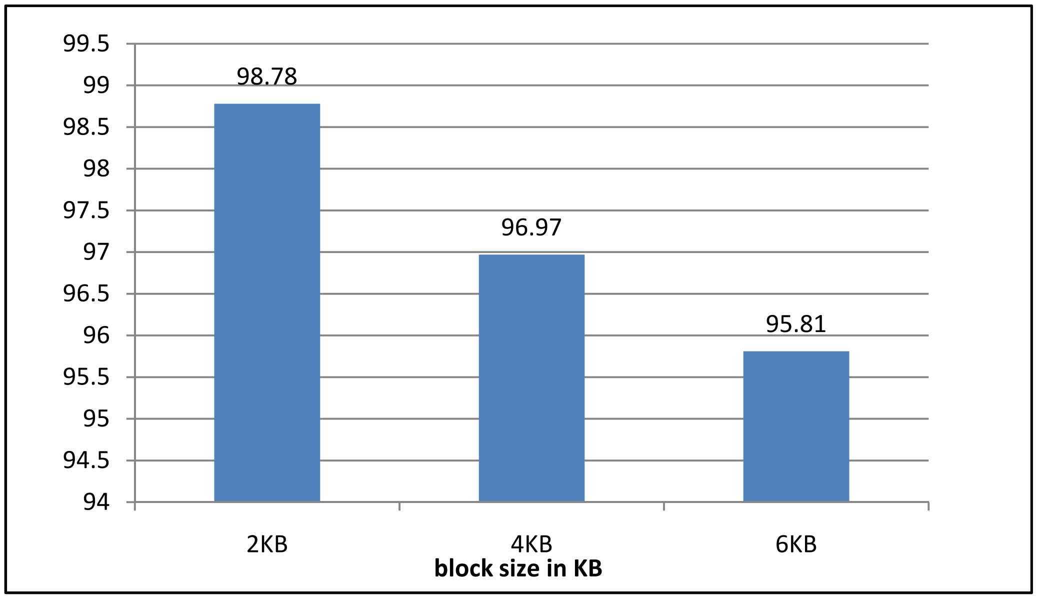


Figure 4.10: the file density when saving UWM_GC.xml dataset

4.5.1.2.2. Group B dataset

We have chosen "**UWM_GB.xml**"; its size is 5.00 MB with 21 distinct elements and 16 distinct paths. The number of disk blocks when the size of the block is 2 KB is 575, the number of disk blocks when the size of the block is 4 KB is 295 and the number of disk blocks when the size of the block is 6 KB is 198 as shown in the following chart

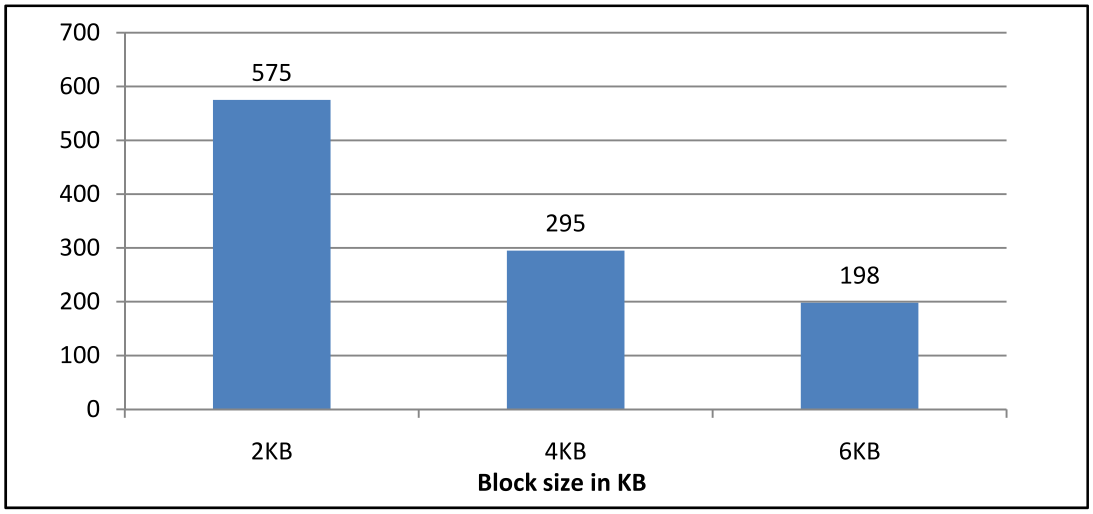


Figure 4.11: the number of blocks used to save UWM_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 98.78%, when the size of the block is 4 KB the density of the file is 96.97 %; and the file density when the size of the block is 6KB is 95.81% as shown in the chart bellow

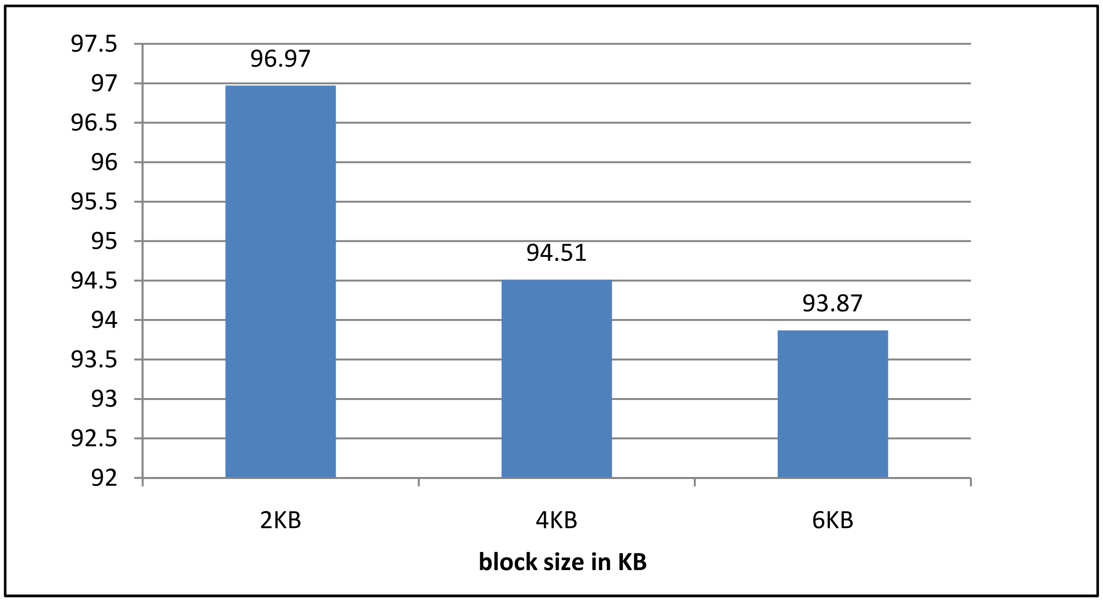


Figure 4.12: the file density when saving UWM_GB.xml dataset

4.5.1.2.3. Group A dataset

We have chosen “**UWM_GA.xml**”; its size is 2.28 MB with 21 distinct elements and 16 distinct paths. The number of disk blocks when the size of the block is 2 KB is 295, the number of disk blocks when the size of the block is 4 KB is 158 and the number of disk blocks when the size of the block is 6 KB is 109 as shown in the following chart

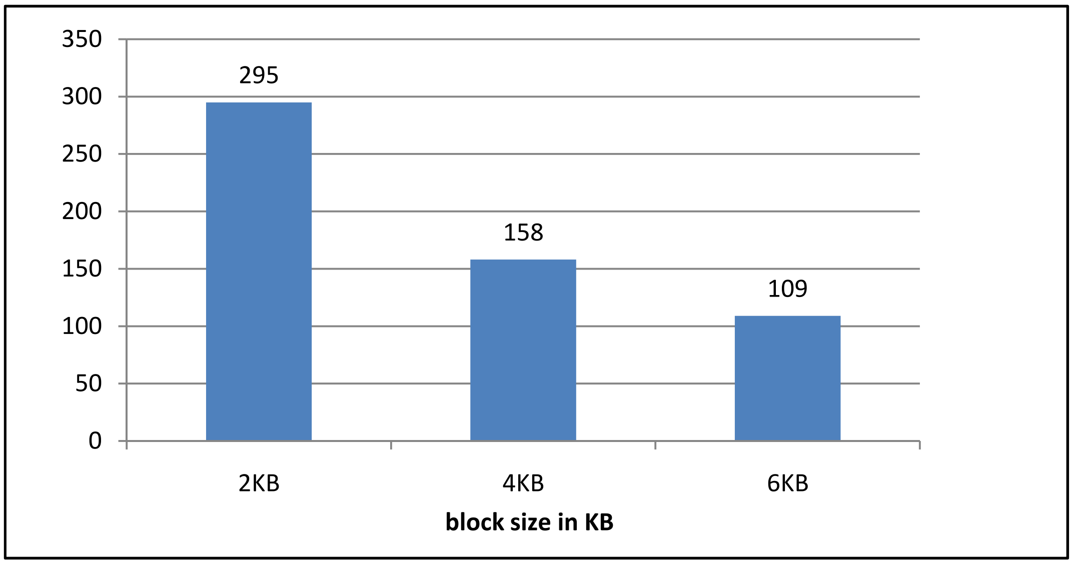


Figure 4.13: the number of blocks used to save UWM_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 94.51%, when the size of the block is 4KB, the density of the file is 88.23 %; and the file density when the size of the block is 6KB is 85.26% as shown in the chart bellow

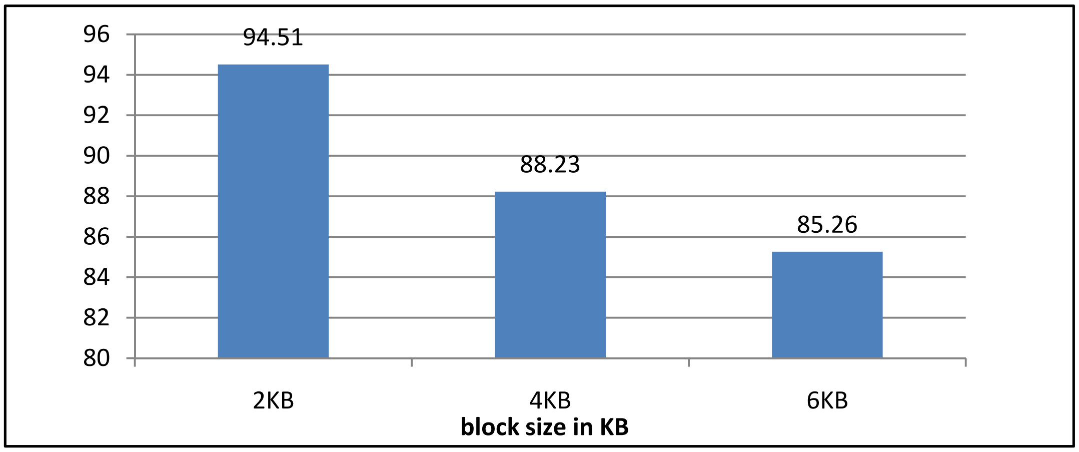


Figure 4.14: the file density when saving UWM_GA.xml dataset

UWM.xml file is has less distinct paths than **SWISSPORT.xml**, so the density of the file becomes larger this in turn, means fewer number of disk blocks will be used to save the data.

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
UWM.xml	9.46	295	158	109	94.51	88.23	85.26
	4.73	575	295	198	96.97	94.51	93.87
	2.28	1129	575	388	98.78	96.97	95.81

Table 4.4: a summery table shows the density file and the number of blocks used when saving UWM.xml

The density of the file in the above table is shown in the following figure

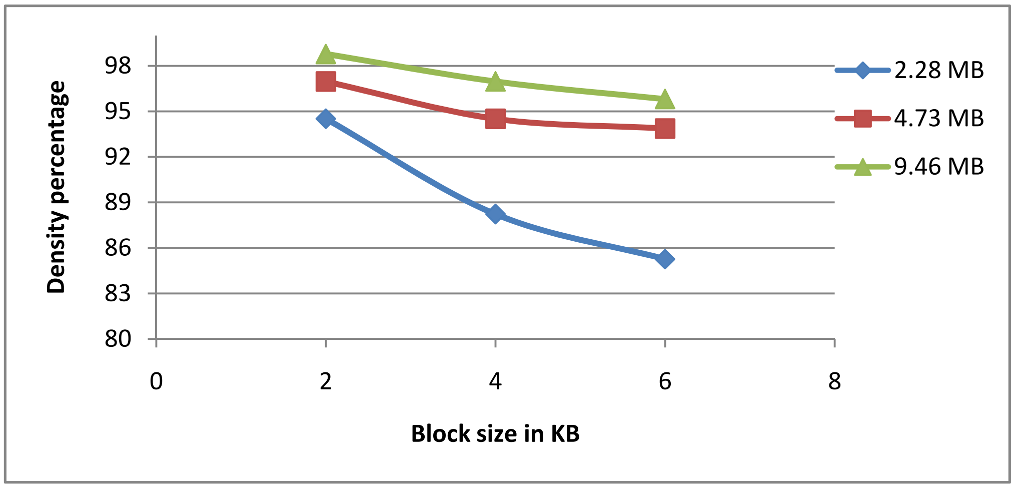


Figure 4.15 the density of MXF when saving UWM.xml datasets

According to the response time of the queries applied on the three versions of the "**UWM.xml**" database, we have applied these simple path queries. These queries are course_listing/note, course_listing/course, course_listing/title, course_listing/credits, course_listing/level,course_listing/restrictions, course_listing/section_listing/section_note and course_listing/section_listing/section as in the following table:

Query number	Number of disk blocks used		
	2kb	4 KB	6 KB
1	38	20	13
2	38	20	13
3	99	50	34
4	99	50	34
5	117	59	40
6	117	59	40
7	36	18	13
8	1	1	1

Table 4.5: shows the query response time for the query set mentioned above

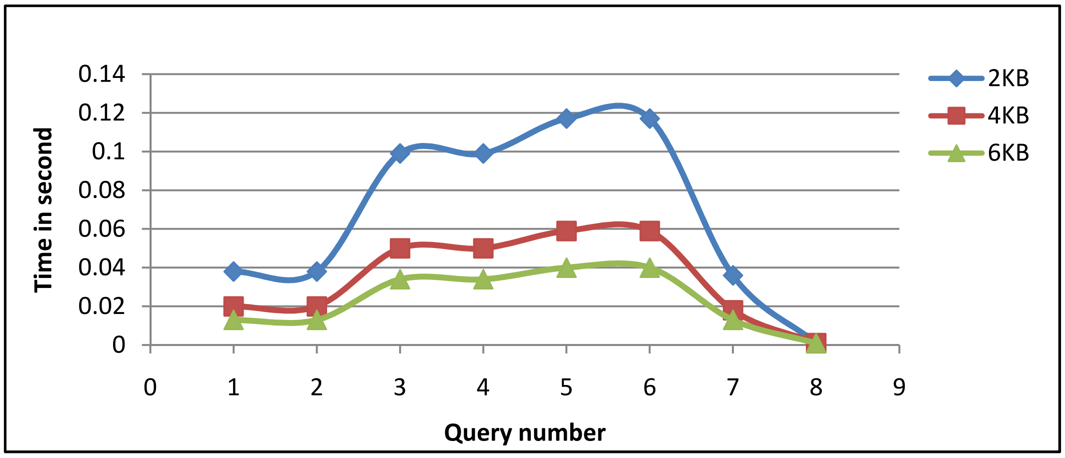


Figure 4.16: query response time when querying UWM.xml datasets

4.5.1.3. Set C datasets

Three groups of datasets will be implemented in this set:

4.5.1.3.1. Group A dataset

We have chosen "**PARTSUPP_GA.xml**" database as; its size is 10.94 MB with 7 distinct elements and 5 distinct paths 5. The number of disk blocks when the size of the block is 2 KB is 885; the number of disk blocks when the size of the block is 4 KB is 445 and the number of disk blocks when the size of the block is 6 KB is 300 as shown in the following chart

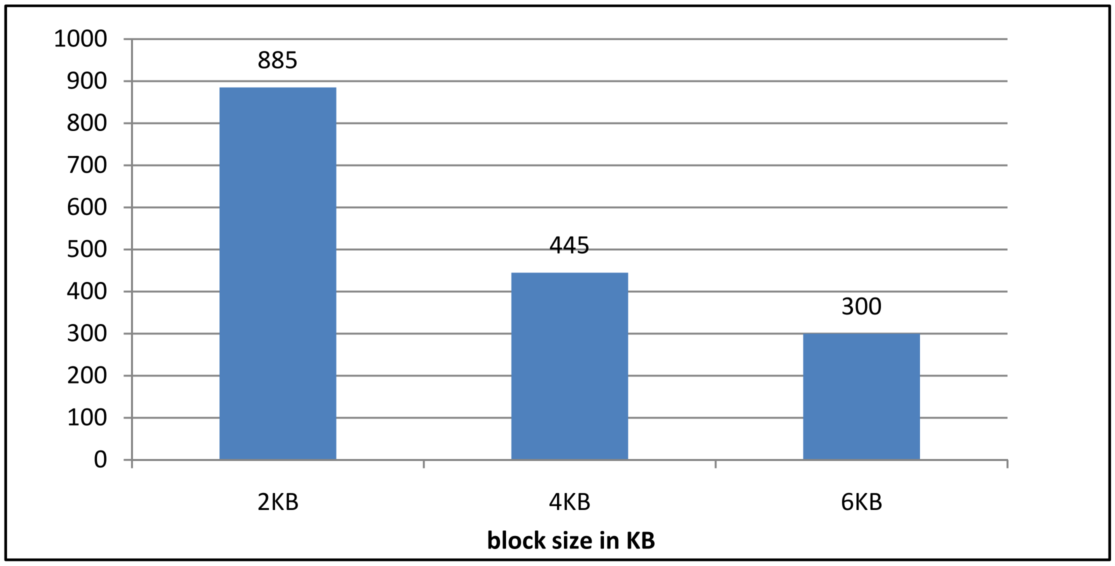


Figure 4.17: the number of blocks used to save PARTSUPP_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 99.31%, when the size of the block is the density of the file is 98.75 %; and the file density when the size of the block is 6KB is 97.65% as shown in the chart bellow

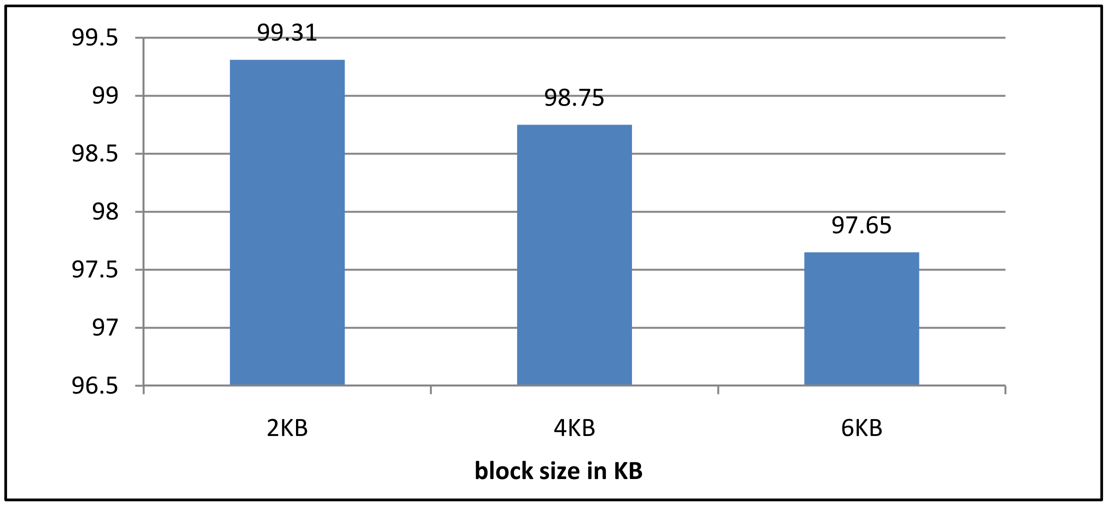


Figure 4.18: the file density when saving PARTSUPP_GC.xml dataset

4.5.1.3.2. Group B dataset

"**PARTSUPP_GB.xml**" was chosen; its size is 4.37 MB with 7 distinct elements and 5 distinct paths. The number of disk blocks when the size of the block is 2 KB is 885; the number of disk blocks when the size of the block is 4 KB is 445 and the number of disk blocks when the size of the block is 6 KB is 300 as shown in the following chart

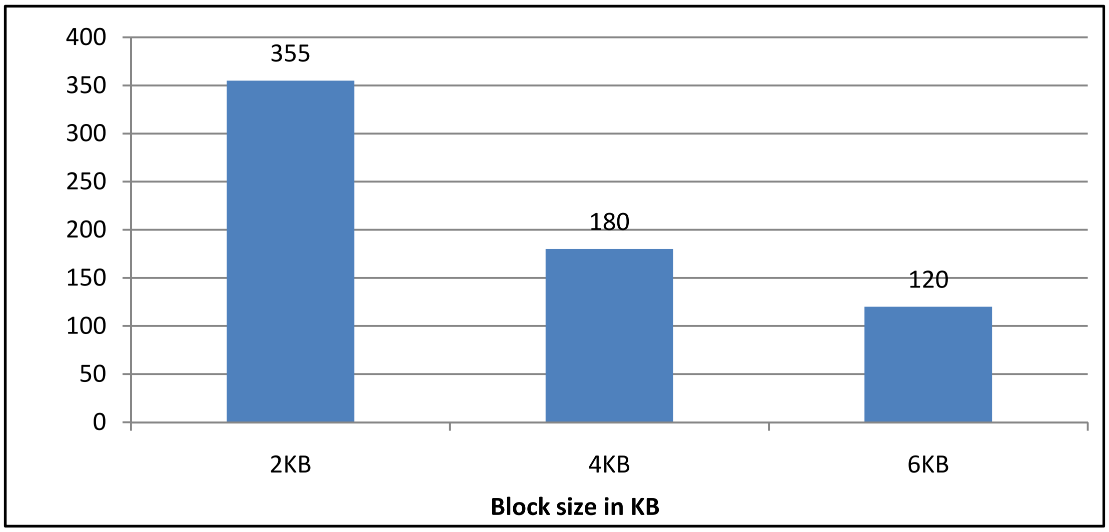


Figure 4.19: the number of blocks used to save PARTSUPP_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 99.03%, when the size of the block is the density of the file is 97.65 %; and the file density when the size of the block is 6KB is 97.65% as shown in the chart bellow

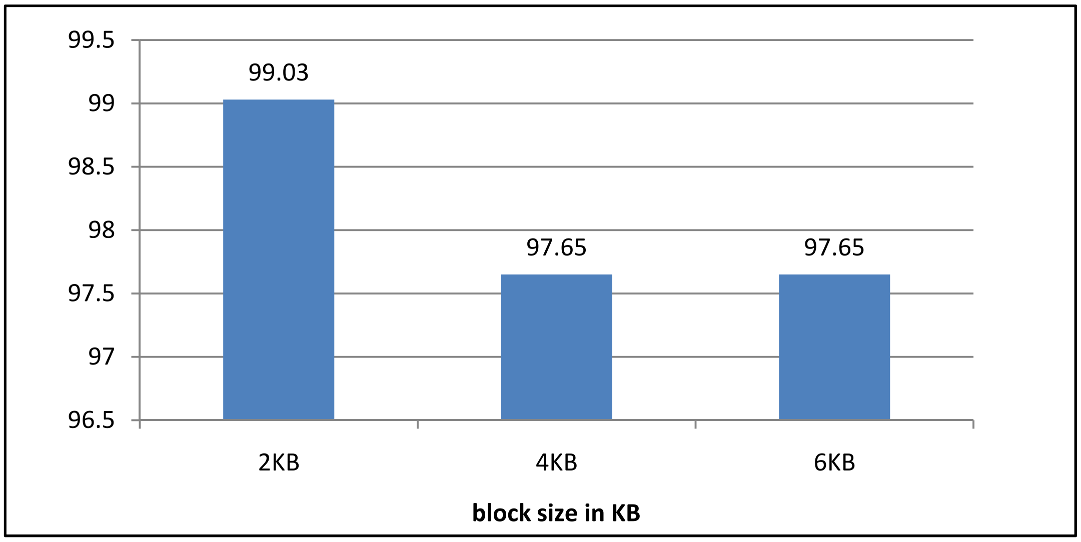


Figure 4.20: the file density when saving PARTSUPP_GB.xml dataset

4.5.1.3.3. Group A dataset

"**PARTSUPP_GA.xml**" was chosen as a dataset with the smallest distinct paths; its size is 2.19 MB with 7 distinct elements and 5 distinct paths. The number of disk blocks when the size of the block is 2 KB is 180, the number of disk blocks when the size of the block is 4 KB is 95 and the number of disk blocks when the size of the block is 6 KB is 65 as shown in the following chart

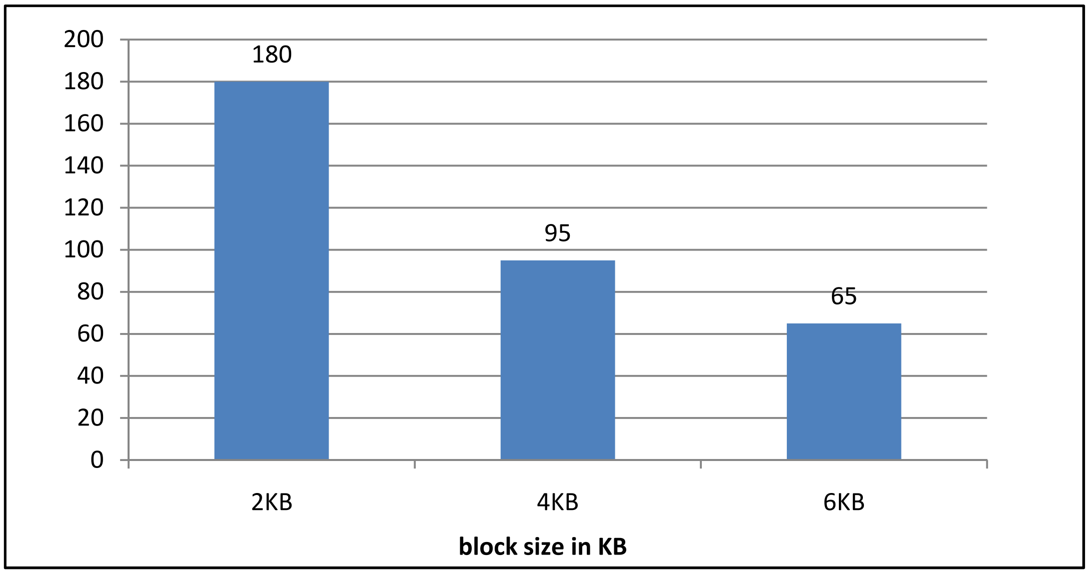


Figure 4.21: the number of blocks used to save PARTSUPP_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 97.65%, when the size of the block is the density of the file is 92.51 %; and the file density when the size of the block is 6KB is 96.14% as shown in the chart bellow

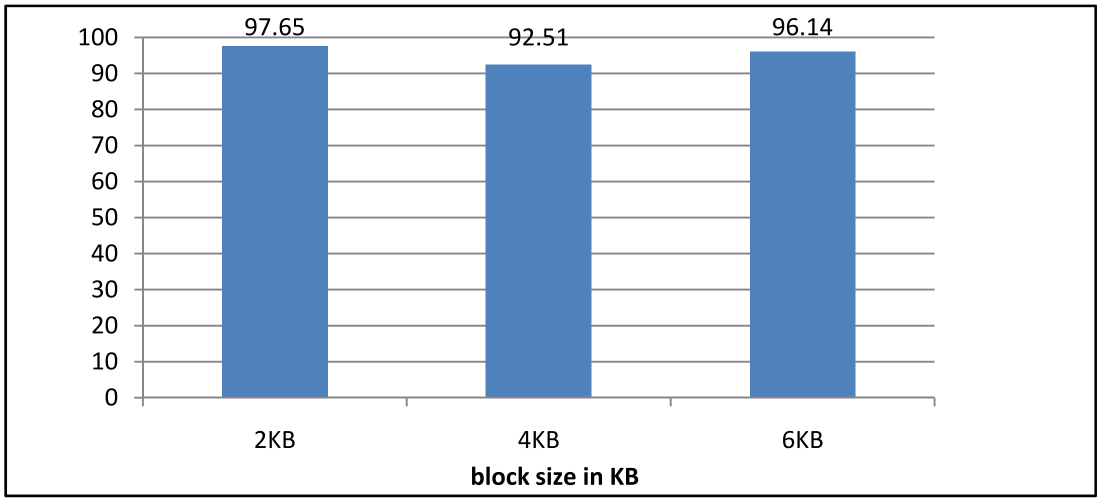


Figure 4.22: the file density when saving PARTSUPP_GC.xml dataset

"**PARTSUPP.xml**" dataset has the smallest number of distinct paths among others, so the density of the file becomes higher than the above two datasets "**SWISSPORT.xml**" and "**UWM.xml**".

Dataset	**Size in MB**	**Number of disk blocks**			**File Density**		
		2 KB	**4 KB**	**6 KB**	**2 KB**	**4 KB**	**6 KB**
PARTSUPP.xml	10.94	885	445	300	99.31	98.75	97.65
	4.37	355	180	120	99.03	97.65	97.65
	2.19	180	95	65	97.65	92.51	96.14

Table 4.6: a summery table shows the density file and the number of blocks used when saving PARTSUPP.xml

The figure bellow shows the density of the file in the above table

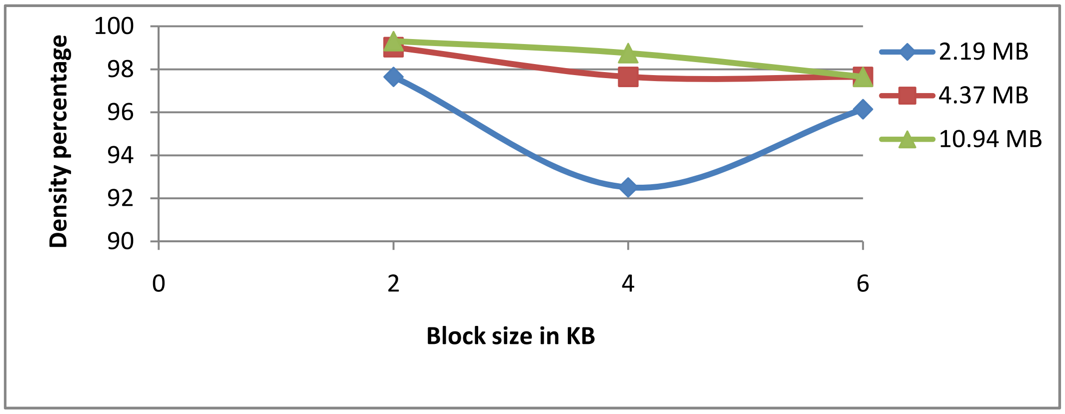


Figure 4.23: the density of the MXF when saving PARTSU.xml datasets

According to the response time of the queries applied on "**PARTSUPP_GA.xml**" database, we have applied these simple path queries. These queries are T/PS_PARTKEY, T/PS_SUPPKEY, T/PS_AVAILQTY, T/PS_SUPPLYCOST, T/PS_SUPPLYCOST and T/PS_COMMENT, as in the following table:

Query number	Number of disk blocks used		
	2kb	4 KB	6 KB
1	71	36	24
2	71	36	24
3	71	36	24
4	71	36	24
5	71	36	24

Table 4.7: shows the query response time for the query set mentioned above

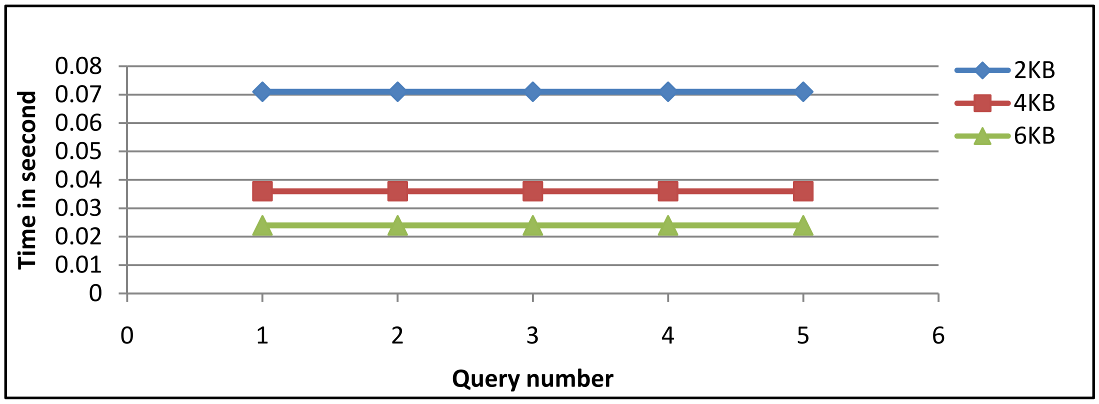


Figure 4.24: the query response time when querying PARTSUPP.xml datasets

4.5.2. Classification the data according to data shape

As we explained in the beginning of this chapter, the shape of the data is width and the depth of the data. The next two sections 5.5.2.1 and 5.5.2.2 explain the behaviour of MXF with the shape of the data.

4.5.2.1. Data-Width datasets

We have implemented three dataset sets according to data-width (set A, set B and set C)

- **Set A**: the data sets used in this set are: three versions of DBLP.xml (DBLP _GC.xml, DBLP _GB.xml and DBLP _GA.xml). This data is the widest among the three sets.
- **Set B**: the data sets used in this set are: three versions of WSU.xml (WSU _GC.xml, WSU _GB.xml and WSU _GA.xml)
- **Set C**: the data sets used in this set are: three versions of CUSTOMER.xml (CUSTOMER _GC.xml, CUSTOMER _GB.xml and CUSTOMER _GA.xml). This set has the smallest width among the three sets.

4.5.2.1.1. Set A datasets

Three groups of datasets will be implemented in this group

4.5.2.1.1.1. Group C dataset

We have chosen "**DBLP_GC.xml**" database; its size is 10.76 MB with 29 distinct elements and 52 distinct paths. The number of disk blocks when the size of the block is 2 KB is 885, the number of disk blocks when the size of the block is 4 KB is 445 and the number of disk blocks when the size of the block is 6 KB is 300 as shown in the following chart

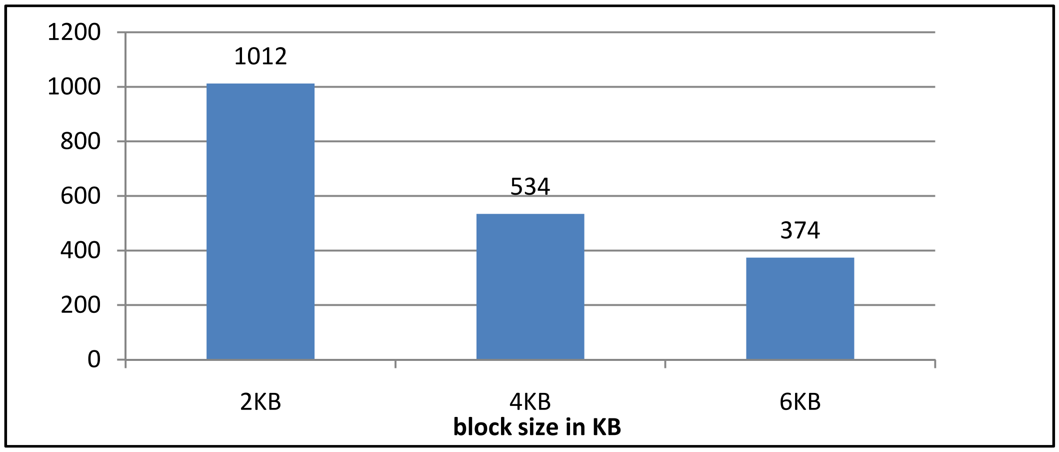


Figure 4.25: the number of blocks used to save DBLP_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 97.65%, when the size of the block is 4KB, the density of the file is 92.51 %; and the file density when the size of the block is 6KB is 90.14% as shown in the chart bellow

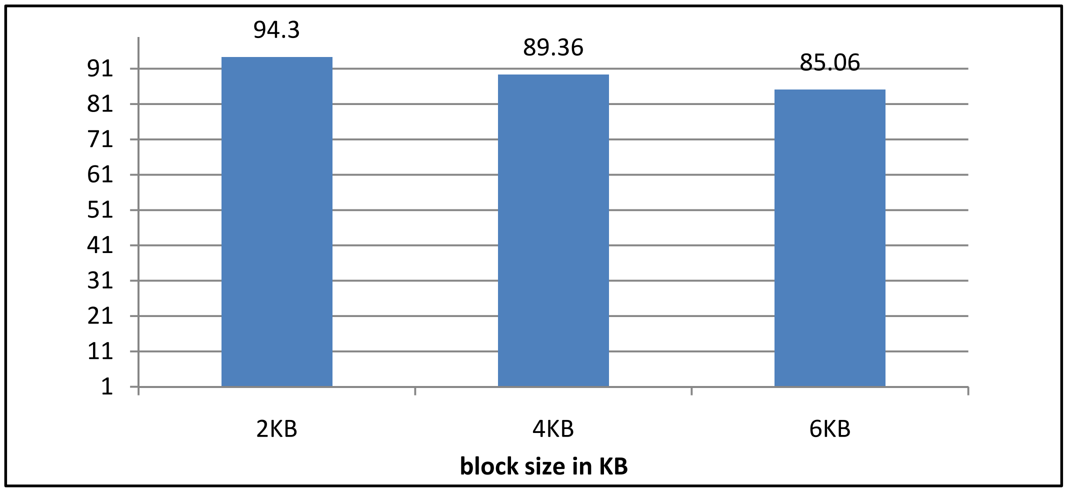


Figure 4.26: the file density when saving DBLP_GC.xml dataset

4.5.2.1.1.2. Group B dataset

We have chosen "**DBLP_GB.xml**" database; its size is 5.74 MB with 29 distinct elements and 52 distinct paths. The number of disk blocks when the size of the block is 2 KB is 578; the number of disk blocks when the size of the block is 4 KB is 315 and the number of disk blocks when the size of the block is 6 KB is 222 as shown in the following chart

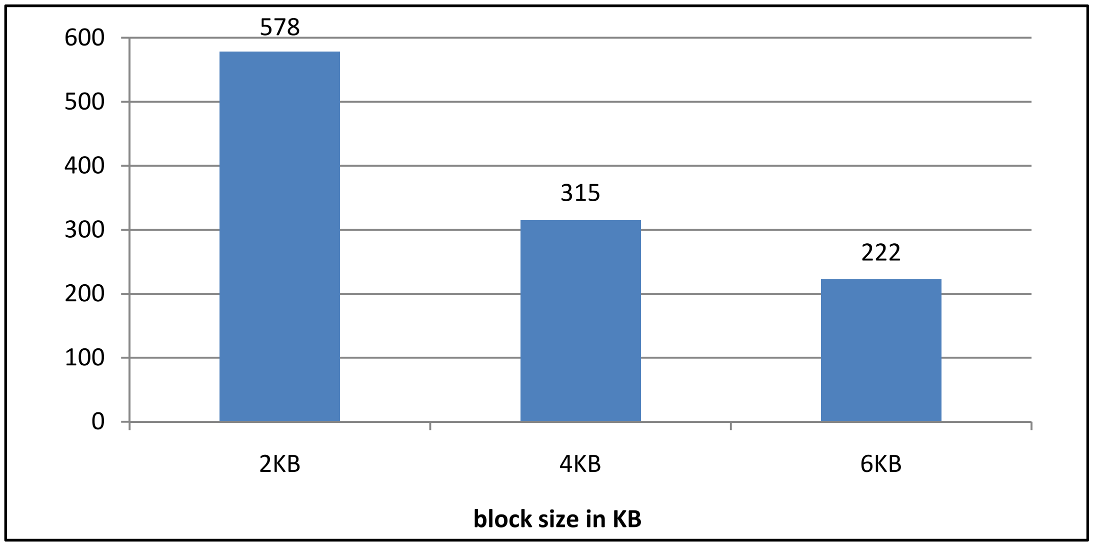


Figure 4.27: the number of blocks used to save DBLP_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 90.86%, when the size of the block is 4KB, the density of the file is 83.36 %; and the file density when the size of the block is 6KB is 78.85% as shown in the chart bellow

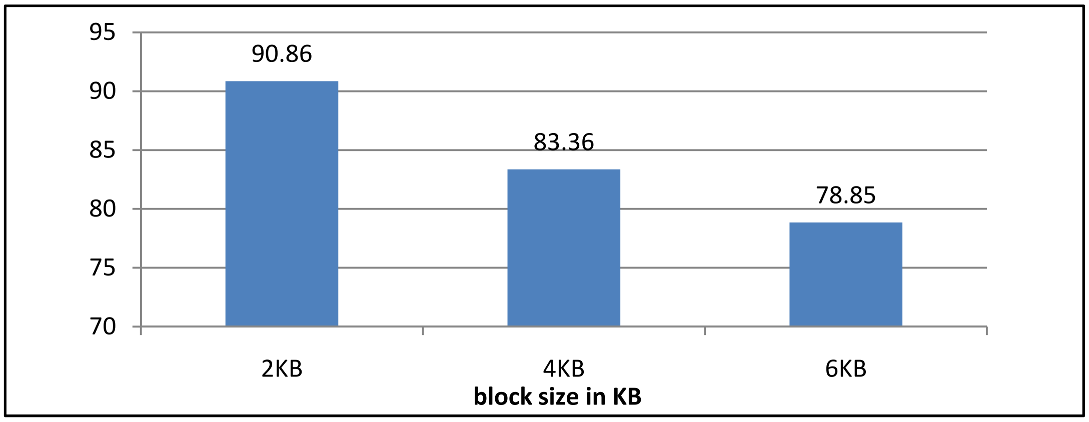


Figure 4.28: the file density when saving DBLP_GB.xml dataset

4.5.2.1.1.3. Group A dataset

We have chosen "**DBLP _GA.xml**" database; its size is 1.36 MB with 29 distinct elements and 52 distinct paths. The number of disk blocks when the size of the block is 2 KB is 176, the number of disk blocks when the size of the block is 4 KB is 114 and the number of disk blocks when the size of the block is 6 KB is 92 as shown in the following chart

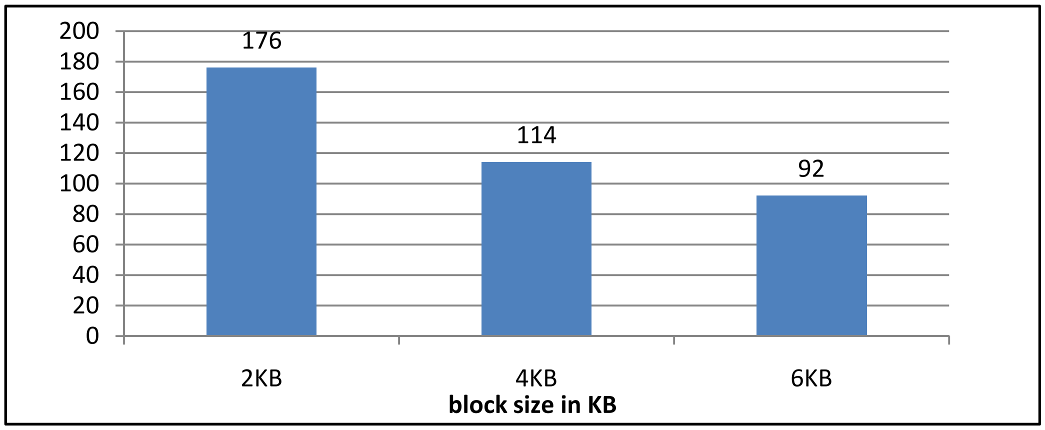


Figure 4.29: the number of blocks used to save DBLP_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 86.86%, when the size of the block is 4KB, the density of the file is 82.36 %; and the file density when the size of the block is 6KB is 72.85% as shown in the chart bellow

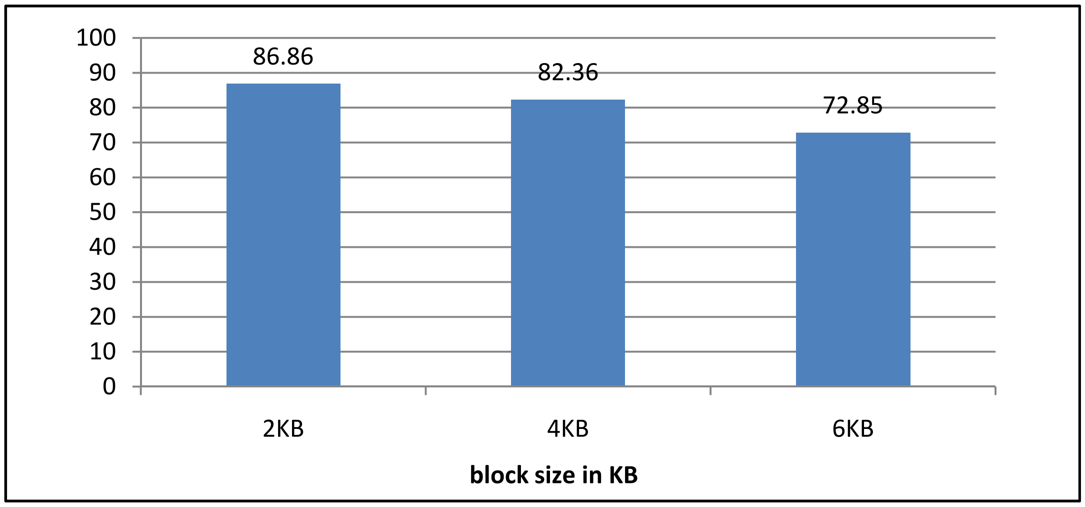


Figure 4.30: the file density when saving DBLP_GA.xml dataset

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
DBLP.xml	10.76	1012	534	374	94.30	89.36	85.06
	5.74	578	315	222	90.86	83.36	78.85
	1.36	176	114	92	86.86	82.36	72.85

Table 4.8: a summery table shows the density file and the number of blocks used when saving DBLP.xml

Data shape of the "**DBLP.xml**" is the widest and from the table above we can conclude that the density of the file is stable regardless of the width of the data but it varies according to the size of the data as shown in the following figure.

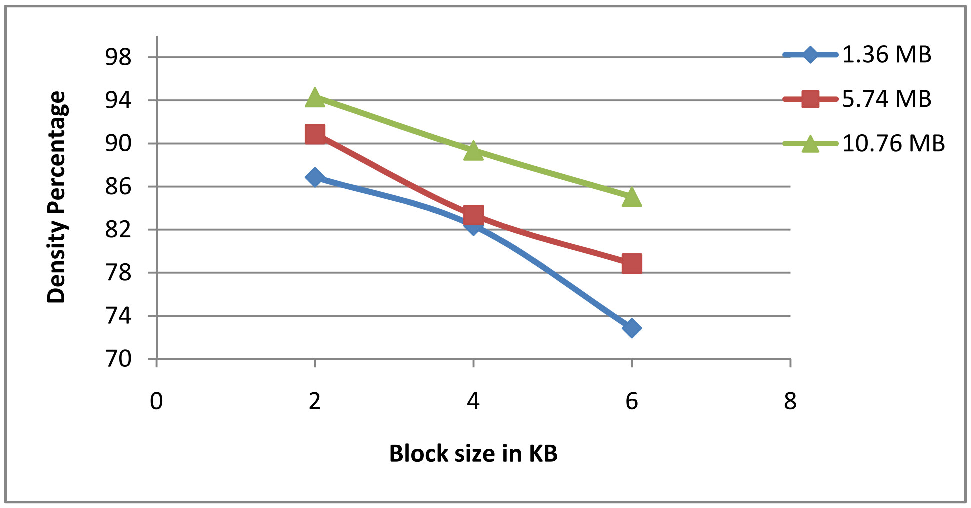


Figure 4.31: the density of MXF when saving DBLP.xml datasets

According to the response time of the queries applied on “DBLP_small.xml” database, we have applied these simple path queries. These queries are mastersthesis/author, article/title, article/ee, article/autho, inproceedings/url, inproceedings/cdrom. Above queries will be numbered in the following table from 1 to 6 as they were ordered above.

Query number	**Number of disk blocks used**		
	2KB	**4 KB**	**6 KB**
1	1	1	1
2	16	8	6
3	10	5	4
4	28	14	10
5	38	19	13
6	7	4	3

Table 4.9: shows the query response time for the query set mentioned above

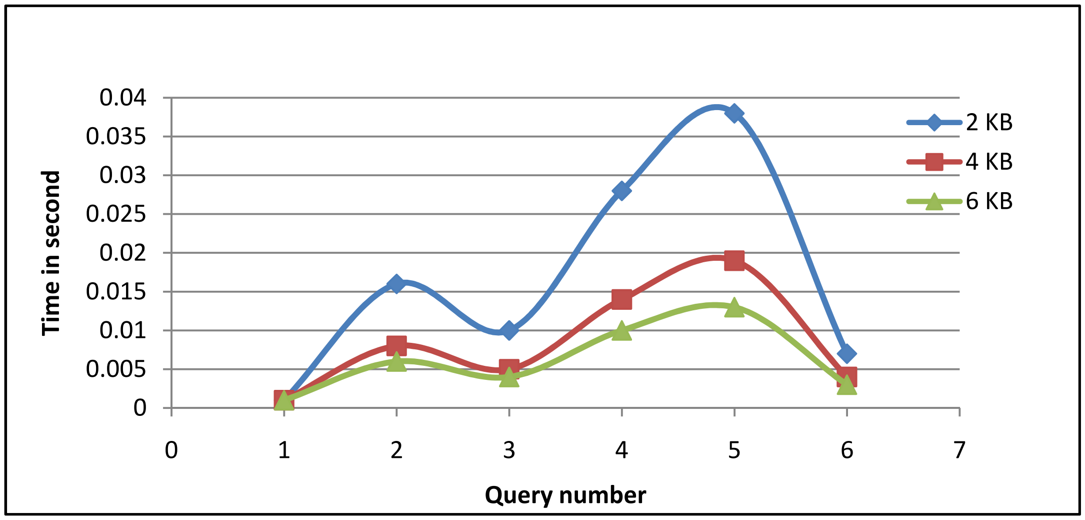


Figure 4.32: the query response time when querying DBLP.xml datasets

4.5.2.1.2. Set B datasets

Three datasets groups will be used in this set

4.5.2.1.2.1. Group C dataset

We have chosen "**WSU_GC.xml**" database; its size is 8.31 MB with 20 distinct elements and 16 distinct paths. The number of disk blocks when the size of the block is 2 KB is 1522, number of disk blocks when the size of the block is 4 KB is 772 and the number of disk blocks when the size of the block is 6 KB is 516 as shown in the following chart

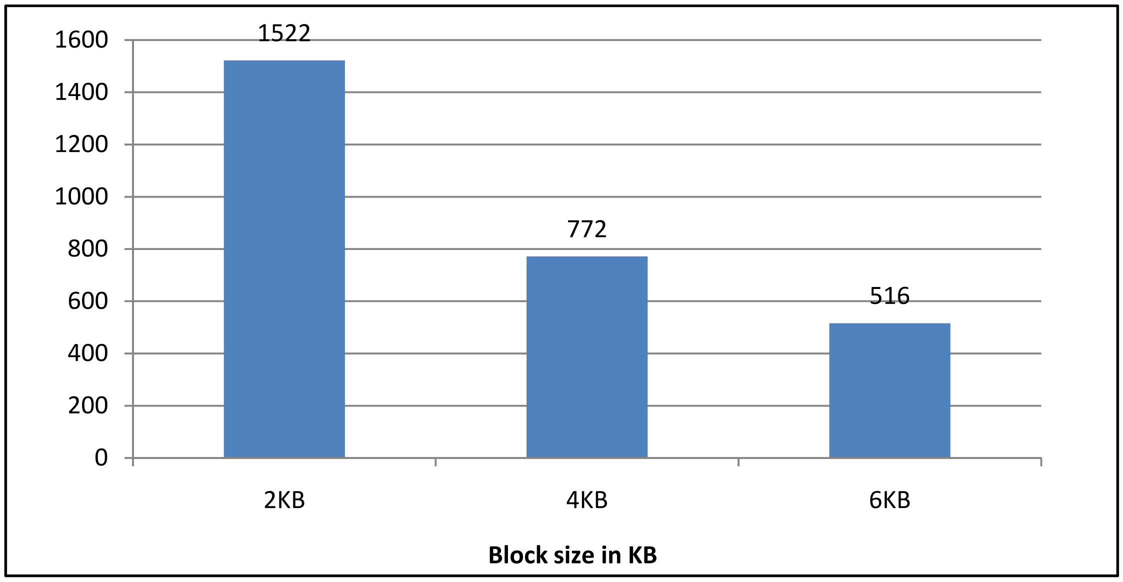


Figure 4.33: the number of blocks used to save WSU_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 74.6%, when the size of the block is 4KB, the density of the file is 57.58 %; and the file density when the size of the block is 6KB is 47.57% as shown in the chart bellow

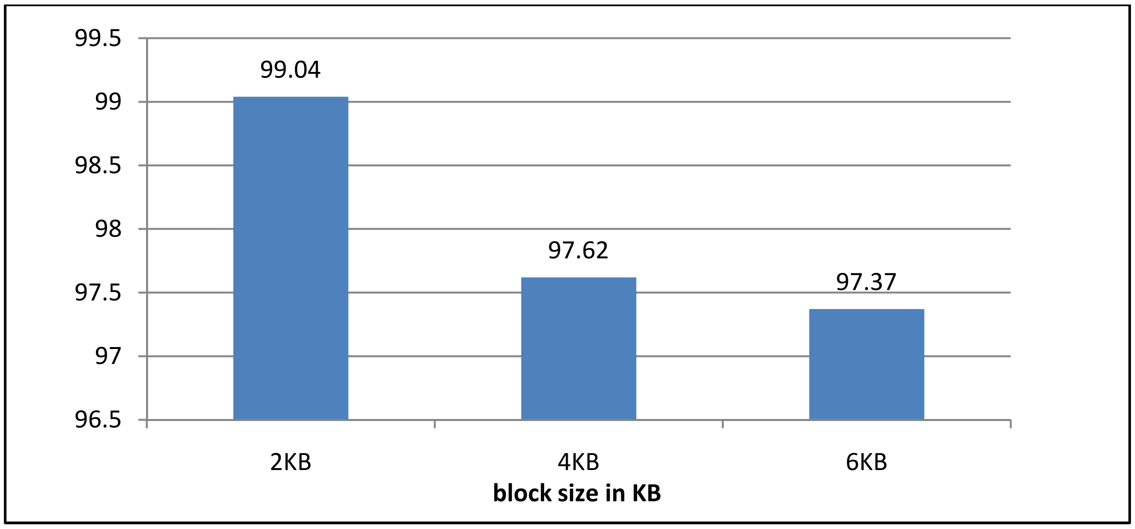


Figure 4.34: the file density when saving WSU_GC.xml dataset

4.5.2.1.2.2. Group B dataset

We have chosen "**WSU_GB.xml**" database; its size is 3.94 MB with 20 distinct elements and 16 distinct paths. The number of disk blocks when the size of the block is 2 KB is 772, the number of disk blocks when the size of the block is 4 KB is 389 and the number of disk blocks when the size of the block is 6 KB is 266 as shown in the following chart

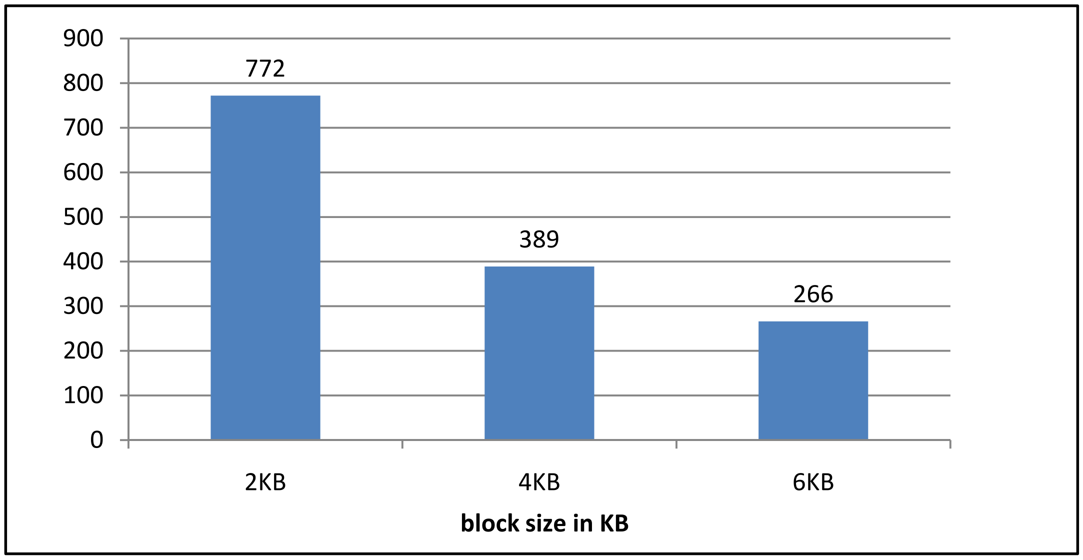


Figure 4.35: the number of blocks used to save WSU_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 97.62%, when the size of the block is 4KB, the density of the file is 96.87 %; and the file density when the size of the block is 6KB is 94.44% as shown in the chart bellow

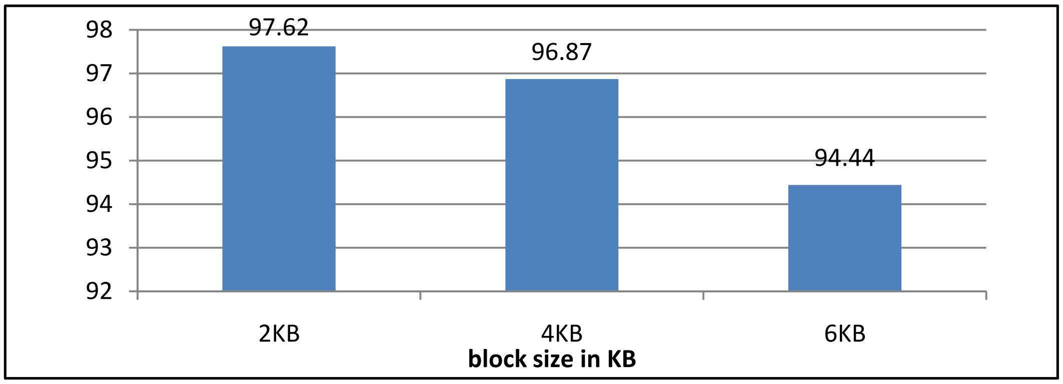


Figure 4.36: the file density when saving DBLP_GB.xml dataset

4.5.2.1.2.3. Group A dataset

We have chosen "**WSU_GA.xml**" database; its size is 1.98 MB with 20 distinct elements and 16 distinct paths. The number of disk blocks when the size of the block is 2 KB is 318, the number of disk blocks when the size of the block is 4 KB is 171 and the number of disk blocks when the size of the block is 6 KB is 120 as shown in the following chart

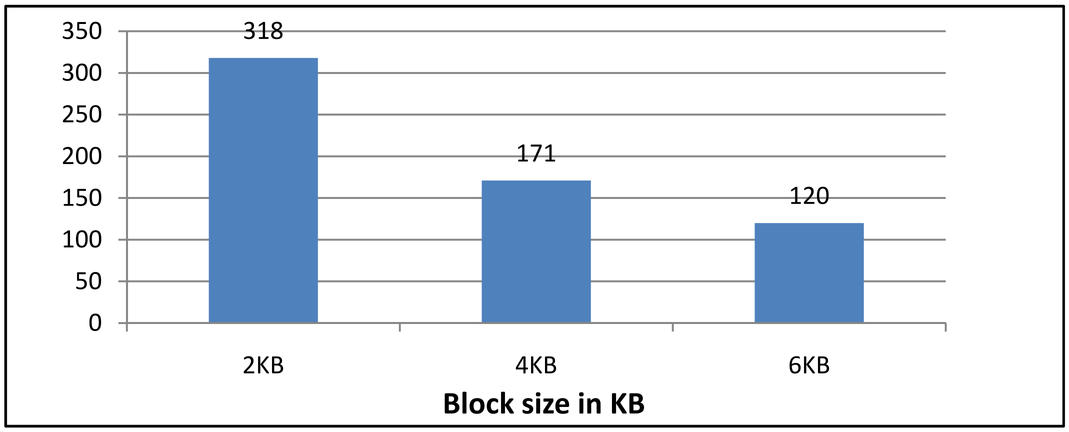


Figure 4.37: the number of blocks used to save DBLP_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 95.8%, when the size of the block is 4KB, the density of the file is 89.07 %; and the file density when the size of the block is 6KB is 84.62% as shown in the chart bellow

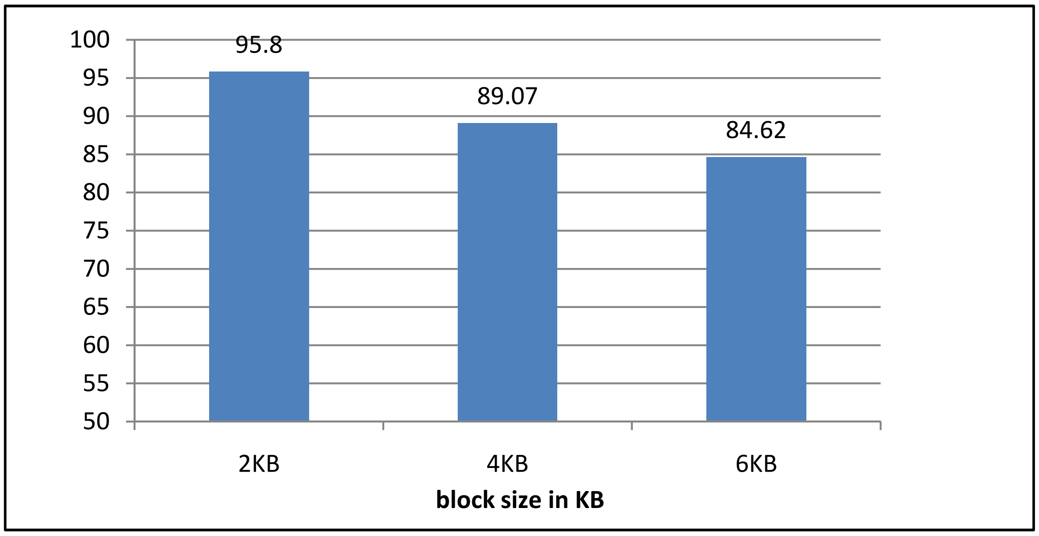


Figure 4.38: the file density when saving DBLP_GA.xml dataset

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
WSU.xml	8.31	1522	722	516	99.04	97.61	97.37
	3.94	722	389	266	97.62	96.87	94.44
	1.98	318	171	120	95.80	89.07	84.62

Table 4.10: a summery table shows the density file and the number of blocks used when saving SWISSPORT.xml

The same observation can be concluded from "**WSU.xml**" results. From the table above we can see that the file density is stable according to the data shape factor but it is affected by the size of the file. But if we compare the result of the "**WSU.xml**" with the results of “**DBLP.xml**” we can conclude that as a data becomes wider, the probability of existing more distinct paths becomes larger; so we see that the overall density of the “**WSU.xml**” dataset is larger than the density of the “**DBLP.xml**” dataset.

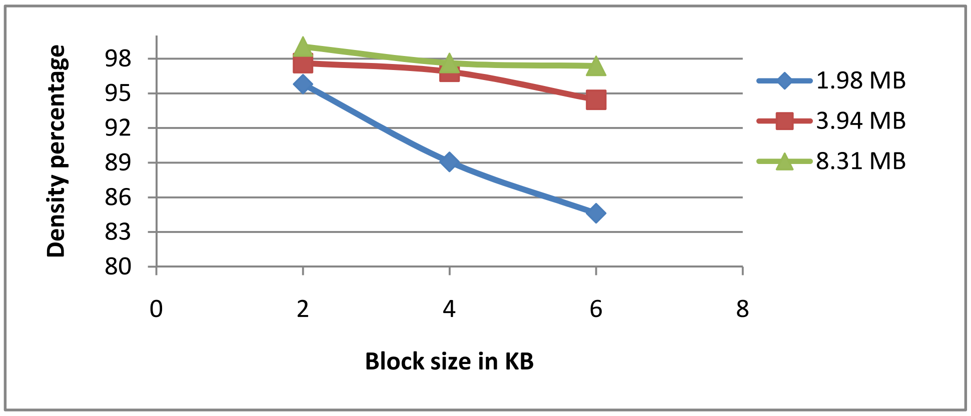


Figure 4.39: the density of MXF when saving WSU.xml datasets

According to the response time of the queries applied on “**WSU_GA.xml**” database, we have applied these simple path queries. These queries are course/footnote, course/sln, course/prefix, course/crs, course/lab, course/sect, course/title and course/credit. These queries will be numbered from 1 to 8 in the following table:

Query number	**Number of disk blocks used**		
	2KB	**4 KB**	**6 KB**
1	44	22	15
2	44	22	15
3	44	22	15
4	44	22	15
5	44	22	15
6	44	22	15
7	44	22	15
8	44	22	15

Table 4.11: shows the query response time for the query set mentioned above

The response time for the set of queries above is illustrated in the following figure:

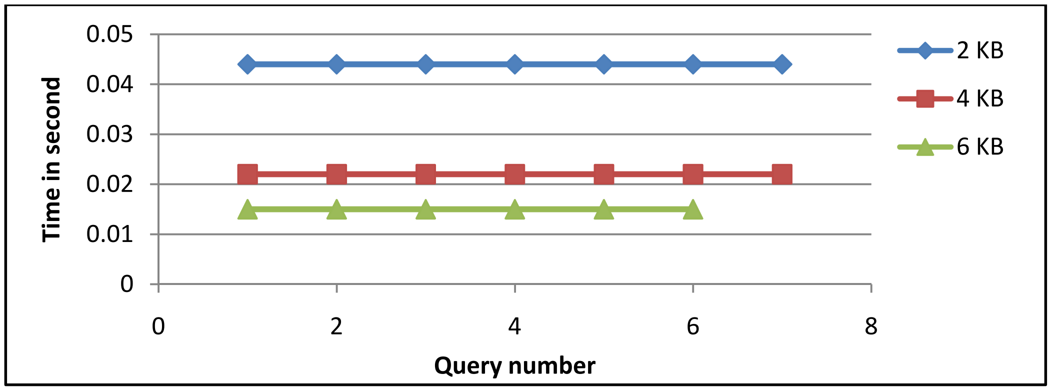


Figure 4.40: the query response time when querying WSU.xml datasets

4.5.2.1.3. Set C datasets

Three datasets groups will be implemented in this set:

4.5.2.1.3.1. Group C dataset

We have chosen "**CUSTOMER_GC.xml**" database; its size is 10.07 MB with 10 distinct elements and 8 distinct paths. The number of disk blocks when the size of the block is 2 KB is 536; the number of disk blocks when the size of the block is 4 KB is 272 and the number of disk blocks when the size of the block is 6 KB is 184 as shown in the following chart

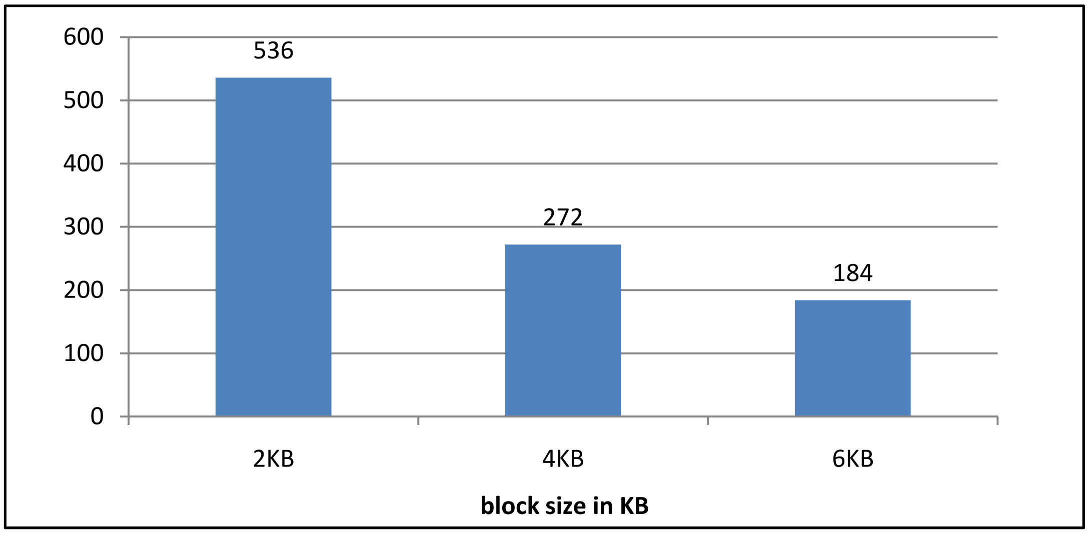


Figure 4.41: the number of blocks used to save CUSTOMER_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 99.38%, when the size of the block is 4KB, the density of the file is 98.93 %; and the file density when the size of the block is 6KB is 97.53% as shown in the chart bellow

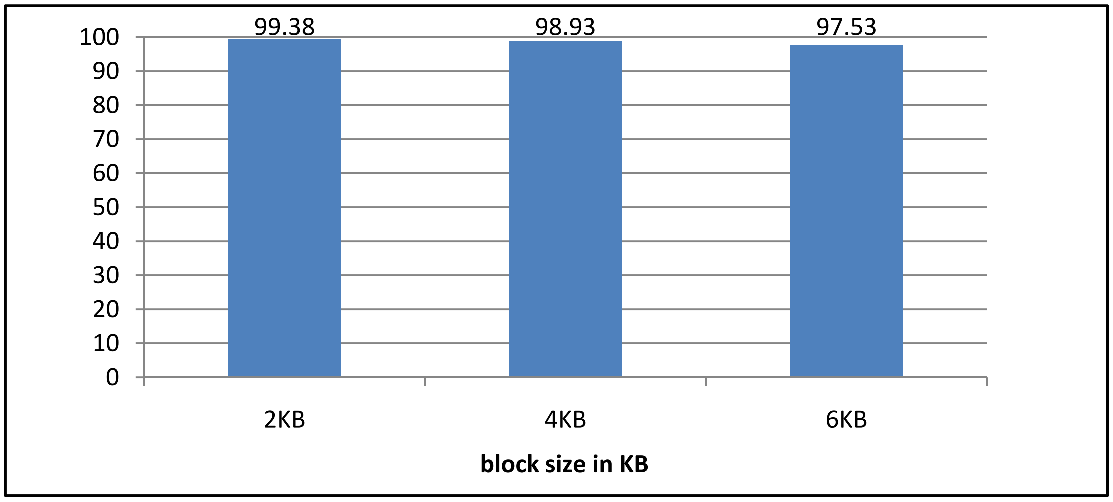


Figure 4.42: the file density when saving CUSTOMER _GC.xml dataset

4.5.2.1.3.2. Group B dataset

As a dataset with a very width shape and average size, we have chosen "**CUSTOMER_GB.xml**" database; its size is 5.03 MB with 10 distinct elements and 8 distinct paths. The number of disk block when the size of the block is 2 KB is 1064, the number of disk block when the size of the block is 4 KB is 536 and the number of disk block when the size of the block is 6 KB is 360 as shown in the following chart

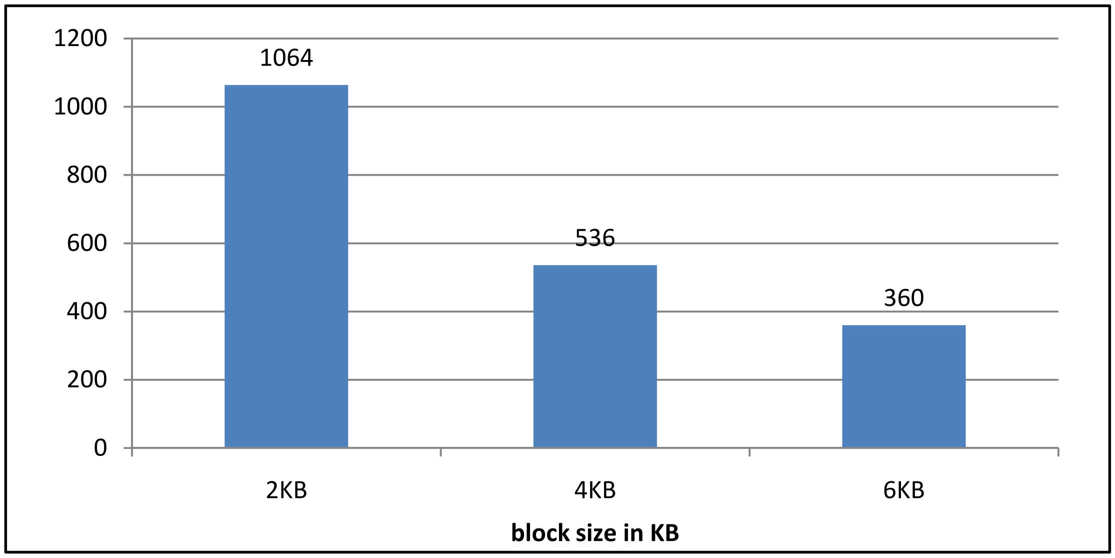


Figure 4.43: the number of blocks used to save CUSTOMER_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 99.12%, when the size of the block is 4KB, the density of the file is 98.38 %; and the file density when the size of the block is 6KB is 97.65% as shown in the chart bellow

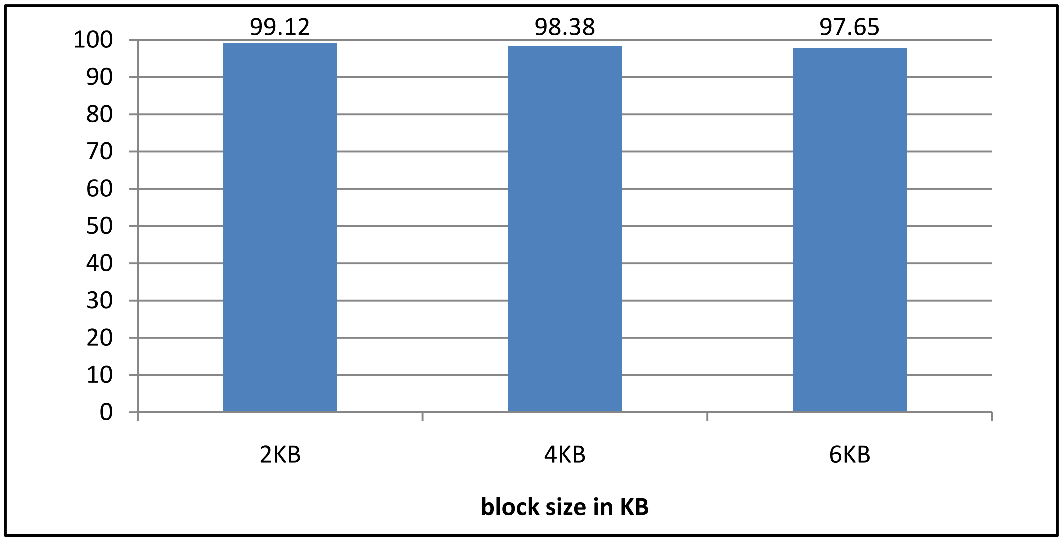


Figure 4.44: the file density when saving CUSTOMER _GB.xml dataset

4.5.2.1.3.3. Group A dataset

We have chosen "**CUSTOMER_GA.xml**" database; its size is 1.51 MB with 8 distinct elements and 10 distinct paths. The number of disk blocks when the size of the block is 2 KB is 64; the number of disk blocks when the size of the block is 4 KB is 32 and the number of disk blocks when the size of the block is 6 KB is 42 as shown in the following chart

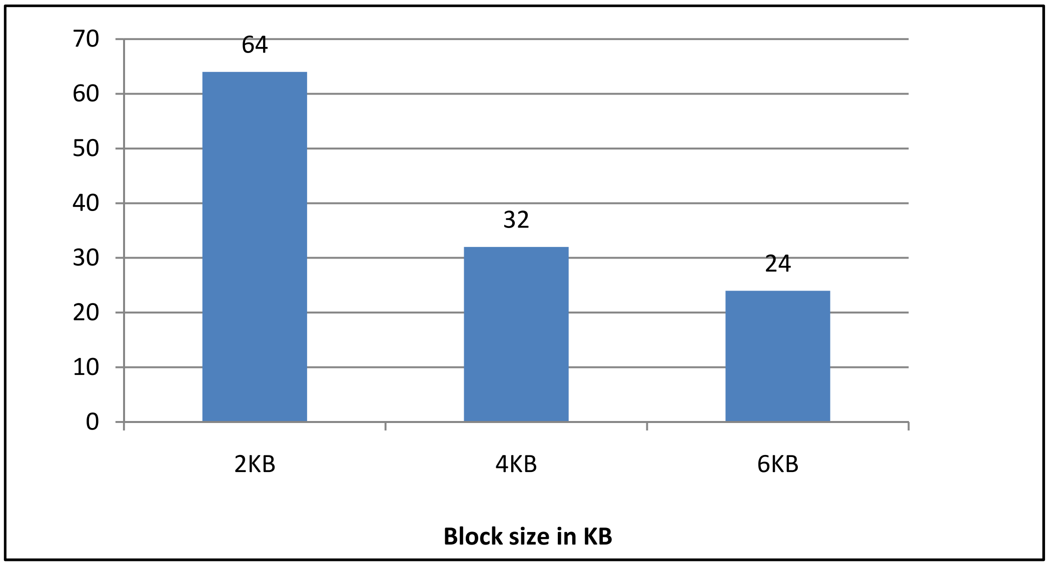


Figure 4.45: the number of blocks used to save CUSTOMER_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 96. 8%, when the size of the block is 4KB, the density of the file is 90.07 %; and the file density when the size of the block is 6KB is 85.62% as shown in the chart bellow

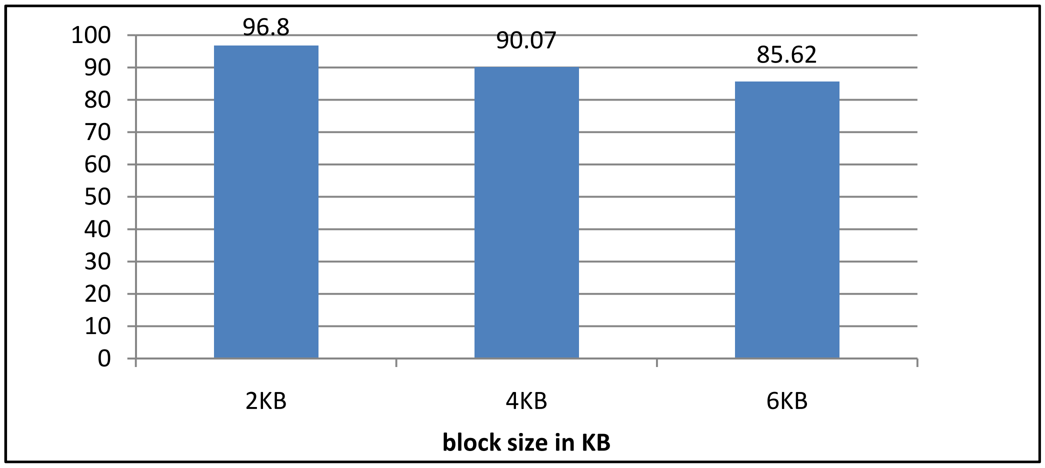


Figure 4.46: the file density when saving CUSTOMER _GA.xml dataset

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
CUSTOMER.xml	10.07	1064	536	360	99.38	98.93	97.53
	5.03	536	272	184	99.12	98.38	97.65
	1.51	64	32	24	96.8	90.07	85.62

Table 4.12: a summery table shows the density file and the number of blocks used when saving CUSTOMER.xml

The same observation can be concluded from "**CUSTOMER.xml**" results. From the table above we can see that the file density is stable according to the data shape factor but it is affected by the size of the file. But if we compare the result of the "**CUSTOMER.xml**" dataset with the two datasets "**WSU.xml**" and "**DBLP.xml**", we can conclude that as a data becomes wider, the probability of existing more distinct paths becomes larger; so we see that the overall density of the "**CUSTOMER.xml**" dataset is larger than the density of the "**WSU.xml**" and "**DBLP.xml**" datasets as shown in the following figure

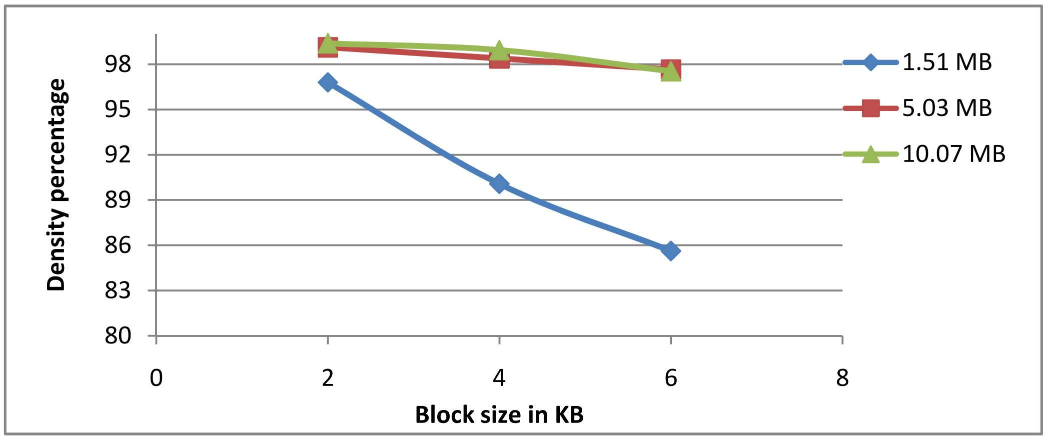


Figure 4. 47: the density of MXF when saving CUSTOMER.xml datasets

According to the response time of the queries applied on "**CUSTOMER_GA.xml**" database, we have applied these simple path queries. These queries are T/C_CUSTKEY, T/C_NAME, T/C_ADDRESS, T/C_NATIONKEY, T/C_PHONE, T/C_ACCTBAL, T/C_MKTSEGMENT and T/C_COMMENT. The figure bellow shows the response time for the query set above where each query has a number from 1 to 8 respectively:

Query number	**Number of disk blocks used**		
	2KB	**4 KB**	**6 KB**
1	67	34	23
2	67	34	23
3	67	34	23
4	67	34	23
5	67	34	23
6	67	34	23
7	67	34	23
8	67	34	23

Table 4.13: shows the query response time for the query set mentioned above.

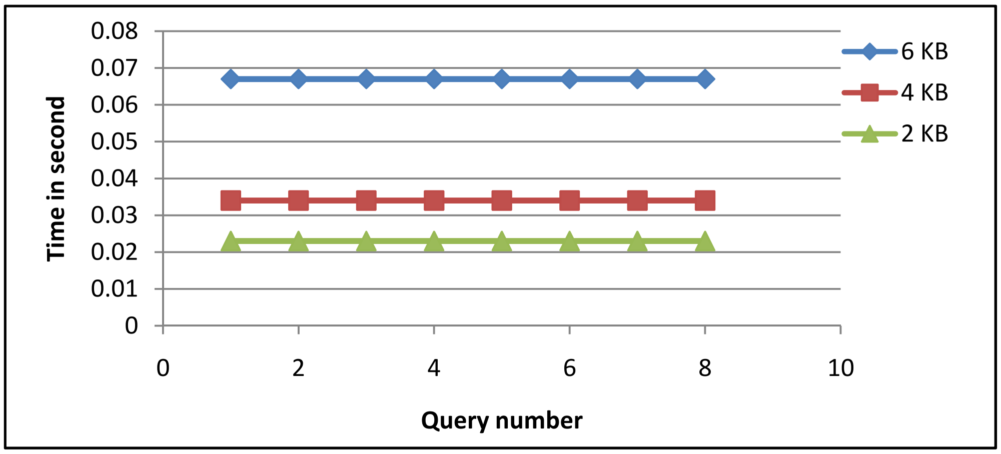


Figure 4.48: the query response time when querying CUSTOMER.xml datasets

4.5.2.2. Data-depth datasets

We have implemented three dataset sets according to data-depth (set A, set B and set C)

- **Set A**: the data sets used in this set are: three versions of PERSONS.xml (PERSONS _GC.xml, PERSONS _GB.xml and PERSONS _GA.xml). This data is the depthest among the three sets.
- **Set B**: the data sets used in this set are: three versions of SIGMODRECORD.xml (SIGMODRECORD _GC.xml, SIGMODRECORD _GB.xml and SIGMODRECORD _GA.xml)
- **Set C**: the data sets used in this set are: three versions of ORDERS.xml (ORDERS _GC.xml, ORDERS _GB.xml and ORDERS _GA.xml). This set has the smallest depth among the three sets.

4.5.2.2.1. Set A datasets

Three groups will be implemented in this set

4.5.2.2.1.1. Group C dataset

We have chosen "**PERSONS_GC.xml**" database; its size is 8.75 MB with 14 distinct elements and 6 distinct paths. The number of disk blocks when the size of the block is 2 KB is 1166; the number of disk blocks when the size of the block is 4 KB is 586 and the number of disk block when the size of the block is 6 KB is 391 as shown in the following chart

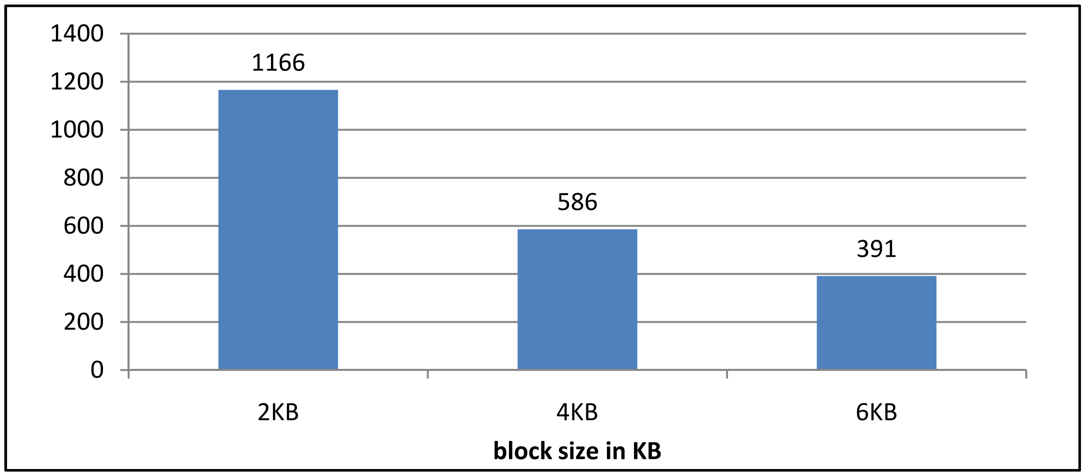


Figure 4.49: the number of blocks used to save PERSONS_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 99.12%, when the size of the block is 4KB, the density of the file is 98.38 %; and the file density when the size of the block is 6KB is 97.65% as shown in the chart bellow

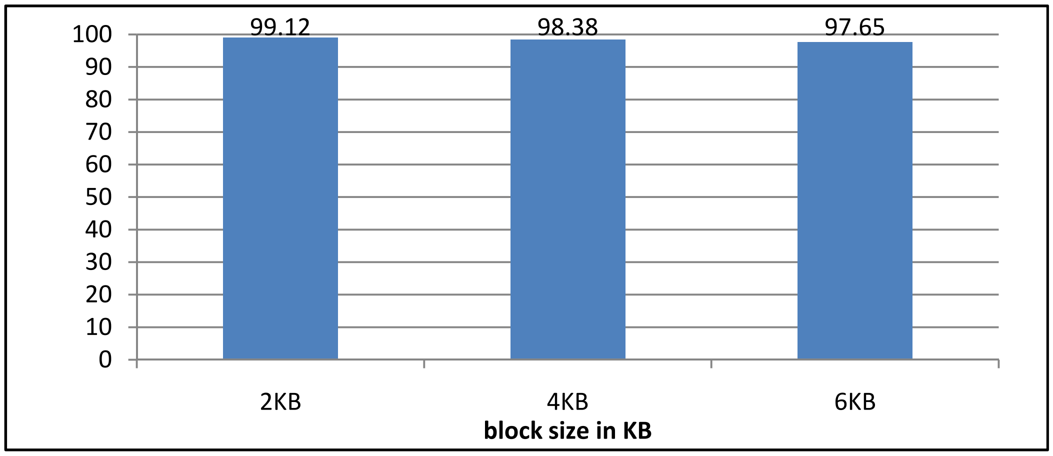


Figure 4.50: the file density when saving PERSONS _GC.xml dataset

4.5.2.2.1.2. Group B dataset

We have chosen "**PERSONS_GB.xml**" database; its size is 4.37 MB with 14 distinct elements and 6 distinct paths. The number of disk blocks when the size of the block is 2 KB is 586; the number of disk blocks when the size of the block is 4 KB is 299 and the number of disk block when the size of the block is 6 KB is 201 as shown in the following chart

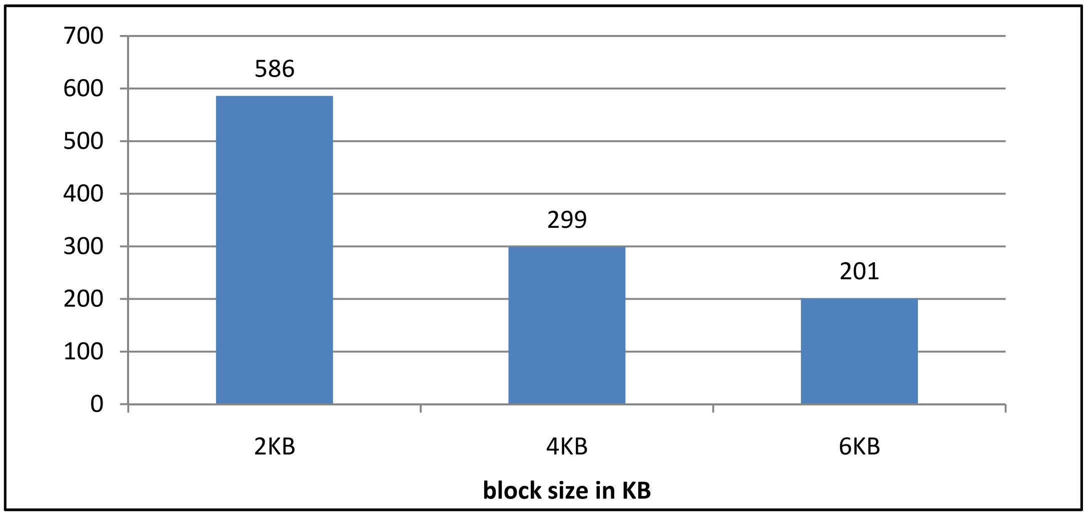


Figure 4.51: the number of blocks used to save PERSONS_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 98.38%, when the size of the block is 4KB, the density of the file is 96.93 %; and the file density when the size of the block is 6KB is 95.53% as shown in the chart bellow

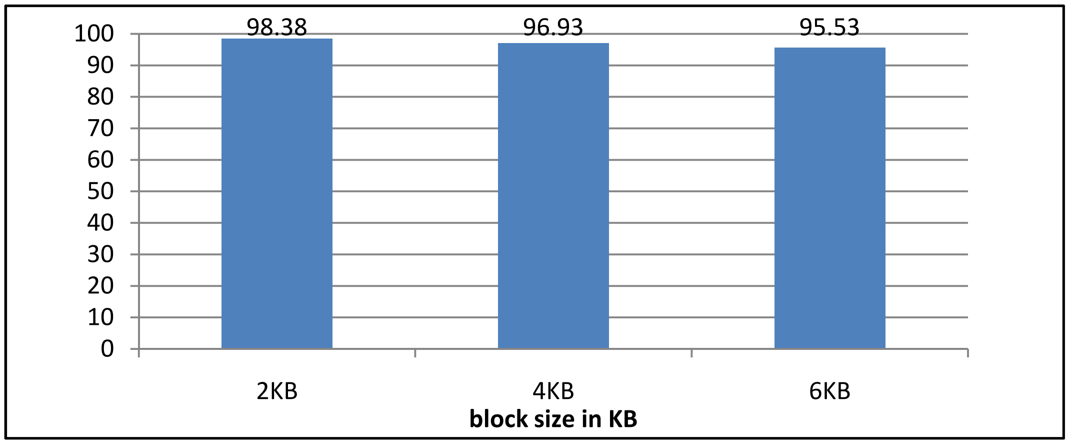


Figure 4.52: the file density when saving PERSONS _GB.xml dataset

4.5.2.2.1.3. Group A dataset

We have chosen "**PERSONS_GA.xml**" database; its size is 1.45 MB with 14 distinct elements and 6 distinct paths. The number of disk block when the size of the block is 2 KB is 201, number of disk block when the size of the block is 4 KB is 104 and the number of disk block when the size of the block is 6 KB is 73 as shown in the following chart

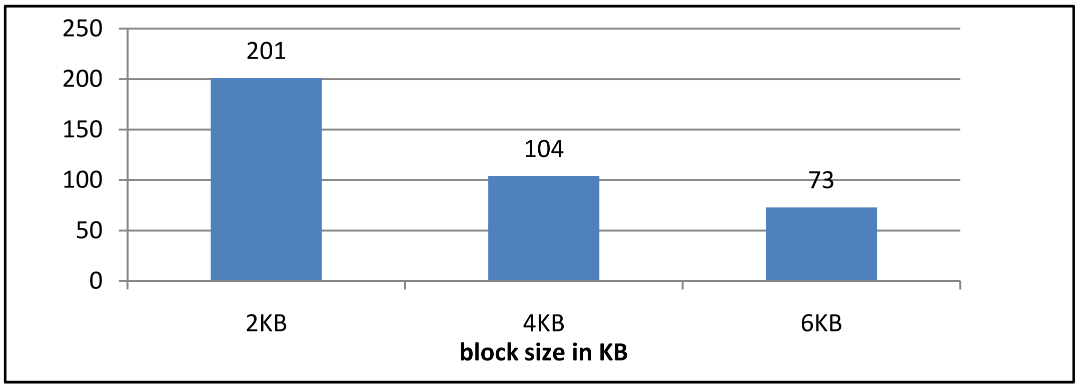


Figure 4.53: the number of blocks used to save PERSONS_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 82.39%, when the size of the block is the density of the file is 82.39 %; and the file density when the size of the block is 6KB is 73.24% as shown in the chart bellow

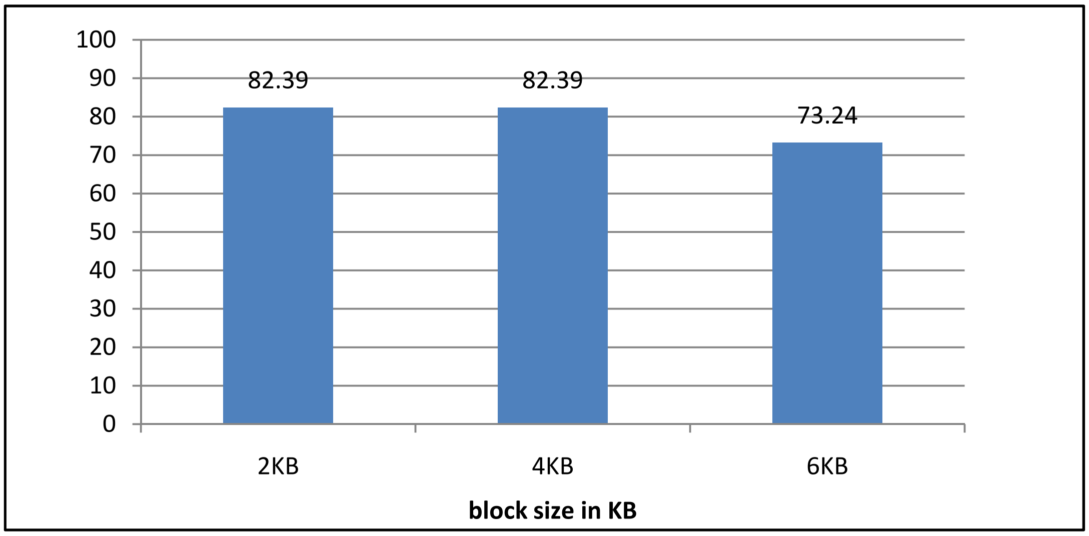


Figure 4.54: the file density when saving PERSONS _GB.xml dataset

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
PERSONS.xml	8.75	1166	586	391	99.12	98.38	97.65
	4.37	586	299	201	98.38	96.93	95.53
	1.45	201	104	73	82.39	82.39	73.24

Table 4.14: a summery table shows the density file and the number of blocks used when saving SWISSPORT.xml

According to the depth factor, since we deal with the path as one unit in an xml data tree, the density of the file will not be affected by the data-depth factor and the size factor will be the dominant factor as shown in the following figure.

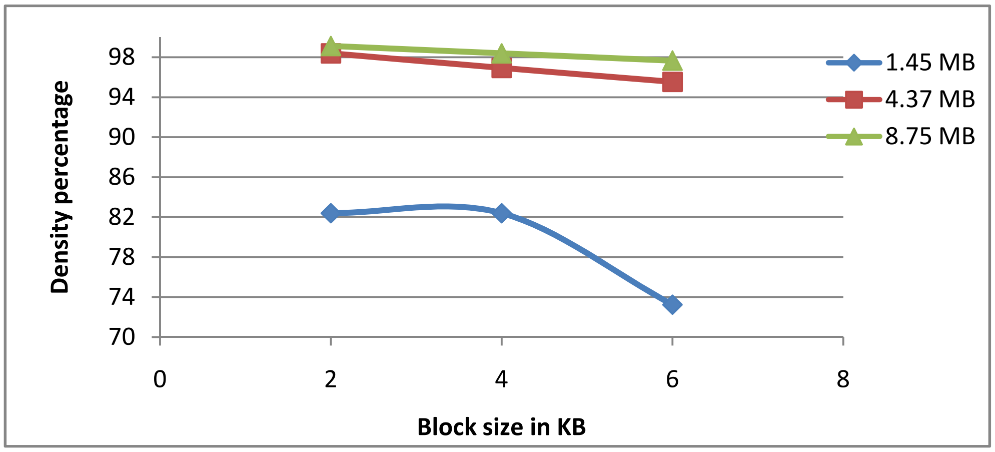


Figure 4.55 the density of MXF when saving PERSONS.xml datasets

According to the response time of the queries applied on "**PERSONS_GA.xml**" database, we have applied these simple path queries. These queries are

1. local_persons/person/person_type/person_info/required_info/special_info/peson_name/person_full
2. local_persons/person/person_type/person_info/required_info/special_info/person_age
3. local_persons/person/person_type/person_info/required_info/special_info/person_religion
4. local_persons/person/person_type/person_info/required_info/special_info/person_state
5. local_persons/person/person_type/person_info/required_info/special_info/person_city
6. local_persons/person/person_type/person_info/required_info/special_info/person_country

Query number	Number of disk blocks used		
	2KB	4 KB	6 KB
1	106	54	36
2	96	49	33
3	90	45	32
4	96	49	33
5	90	45	32
6	96	49	33

Table 4.15: shows the query response time for the query set mentioned above

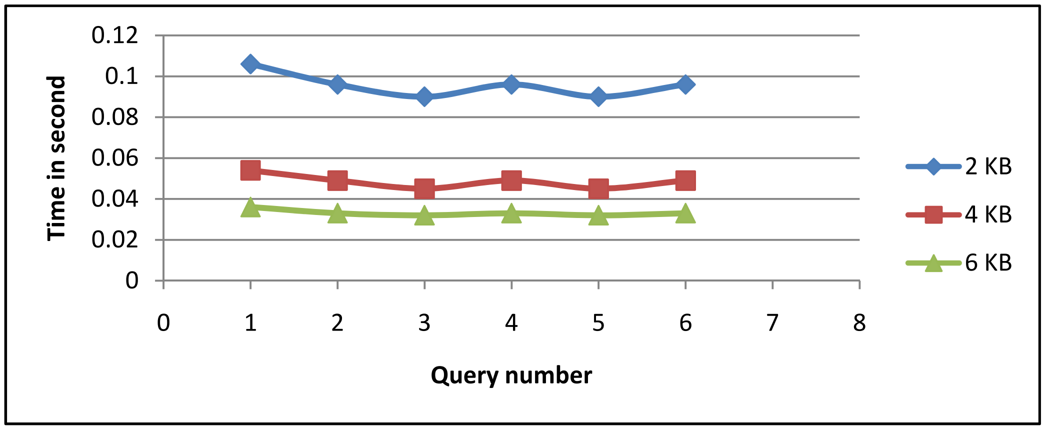


Figure 4.56: the query response time when querying PERSONS.xml datasets

4.5.2.2.2. Set B datasets

Three dataset groups will be implemented in this set:

4.5.2.2.2.1. Group C dataset

We have chosen "**SIGMODRECORD _GC.xml**" database; its size is 13.05 MB with 11 distinct elements and 6 distinct paths. The number of disk blocks when the size of the block is 2 KB is 1535, the number of disk blocks when the size of the block is 4 KB is 771 and the number of disk blocks when the size of the block is 6 KB is 516 as shown in the following chart

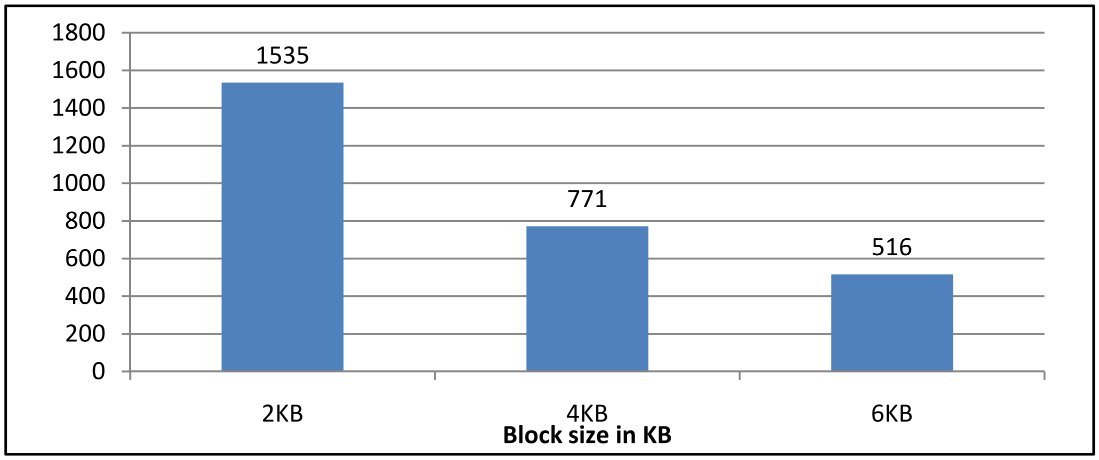


Figure 4.57: the number of blocks used to save SIGMODRECORD_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 99.55%, when the size of the block is the density of the file is 99.1 %; and the file density when the size of the block is 6KB is 98.72% as shown in the chart bellow

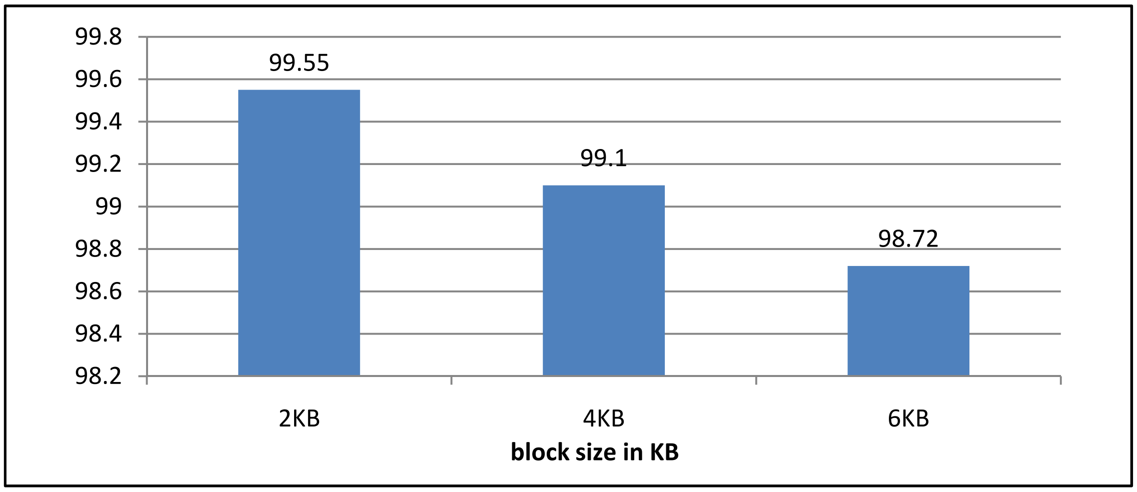


Figure 4.58: the file density when saving SIGMODRECORD _GC.xml dataset

4.5.2.2.2.2. Group B dataset

We have chosen “**SIGMODRECORD _GB.xml**” database; its size is 4.35 MB with 11 distinct elements and 6 distinct paths. The number of disk blocks when the size of the block is 2 KB is 516, the number of disk blocks when the size of the block is 4 KB is 260 and the number of disk blocks when the size of the block is 6 KB is 177 as shown in the following chart

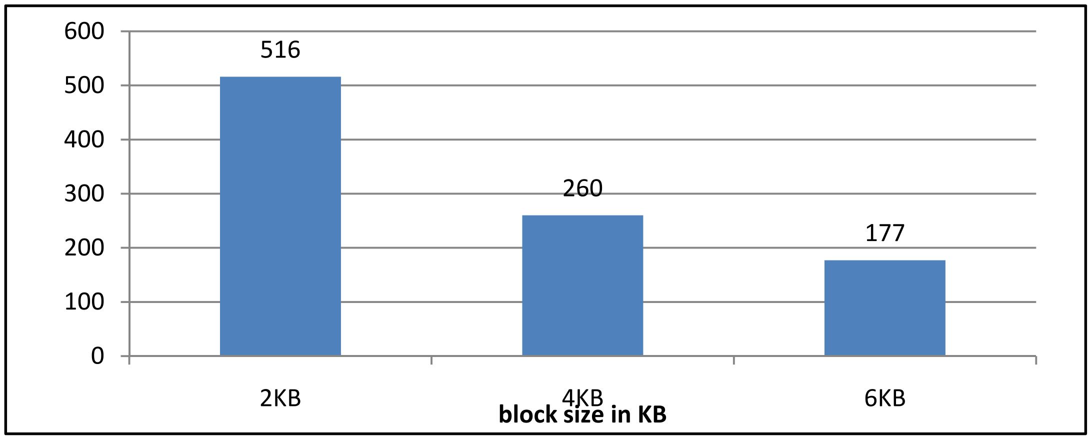


Figure 4.59: the number of blocks used to save SIGMODRECORD_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 98.38%, when the size of the block is 4KB, the density of the file is 96.93 %; and the file density when the size of the block is 6KB is 95.53% as shown in the chart bellow

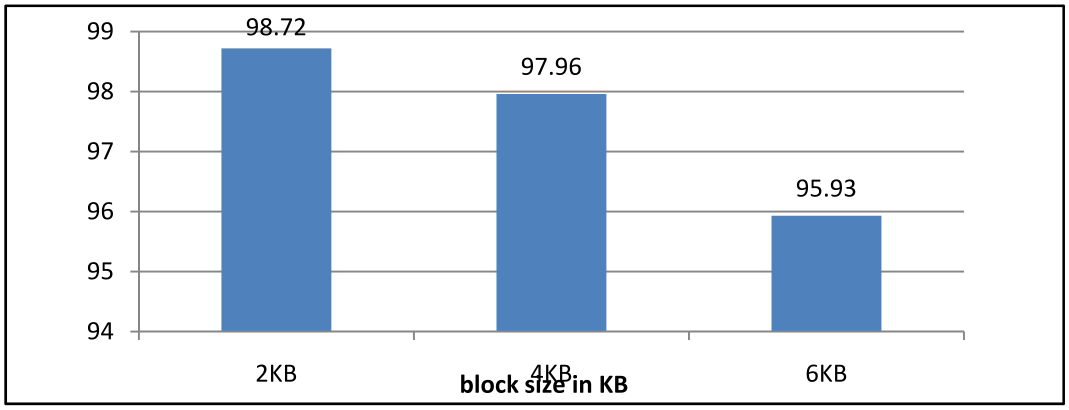


Figure 4.60: the file density when saving SIGMODRECORD _GB.xml dataset

4.5.2.2.2.3. Group A dataset

We have chosen "**SIGMODRECORD_GA.xml**" database; its size is 1.47 MB with 11 distinct elements and 6 distinct paths. The number of disk blocks when the size of the block is 2 KB is 177; the number of disk blocks when the size of the block is 4 KB is 89 and the number of disk blocks when the size of the block is 6 KB is 63 as shown in the following chart

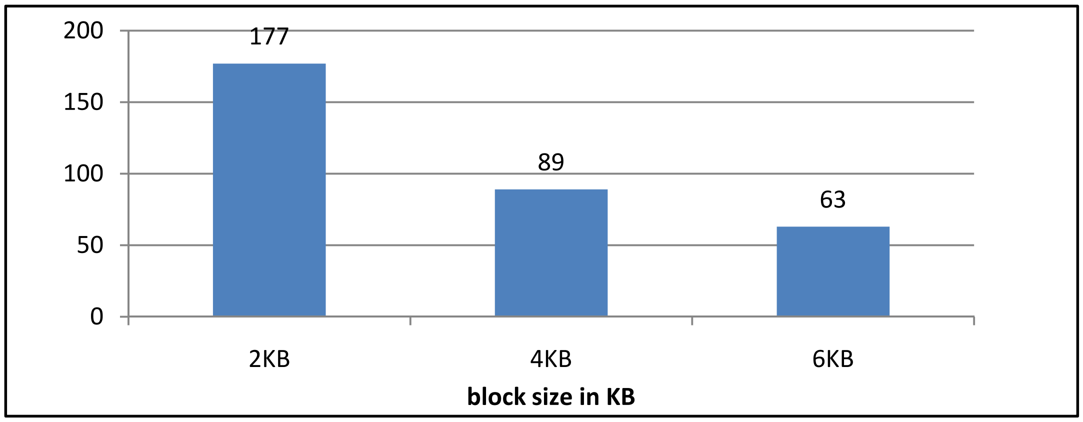


Figure 4.61: the number of blocks used to save SIGMODRECORD_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 95.93%, when the size of the block is 4KB, the density of the file is 95.39 %; and the file density when the size of the block is 6KB is 89.84% as shown in the chart bellow

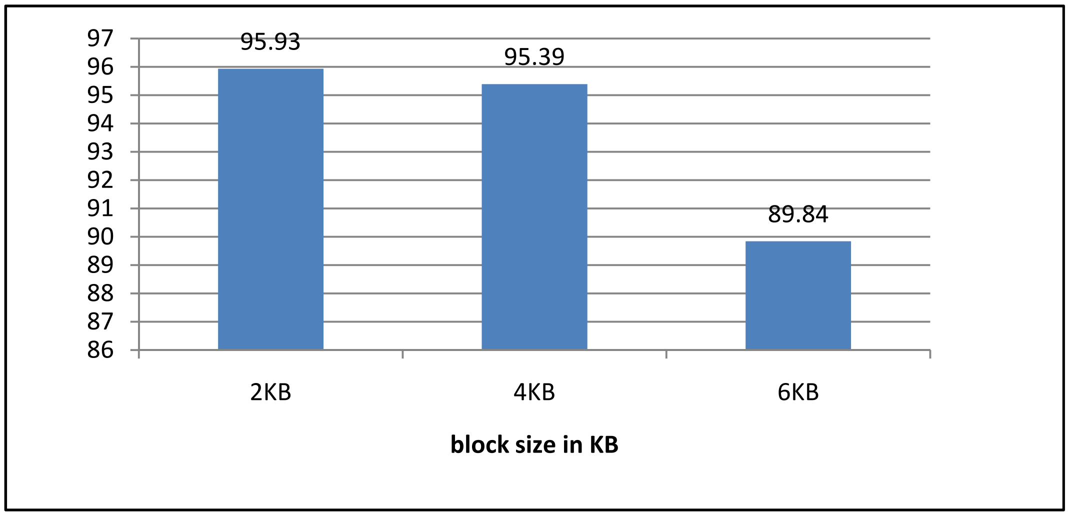


Figure 4.62: the file density when saving SIGMODRECORD _GA.xml dataset

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
SIGMODRECORD.xml	13.05	1535	771	516	99.55	99.10	98.72
	4.35	516	260	177	98.72	97.96	95.93
	1.45	177	89	63	95.93	95.39	89.84

Table 4.16: a summery table shows the density file and the number of blocks used when saving SIGMODRECORD.xml

“**SIGMODRECORD.xml**” has less depth than “**PERSONS.xml**”, but there is a slight difference in the file density which emphasizes that the depth factor will not affect the density of the file or has a slight effect on the density of the file.

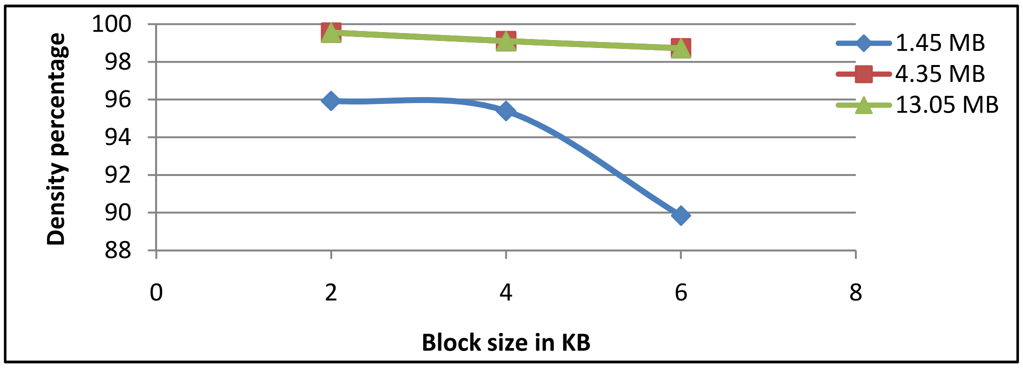


Figure 4.63: the density of MXF when saving SIMORECORD.xml datasets

According to the response time of the queries applied on “**SIGMODRECORD_GA.xml**” database, we have applied these simple path queries. These queries are issue/volume, issue/number, issue/articles/article/title, issue/articles/article/initPage, issue/articles/article/endPage and issue/articles/article/authors/author. Queries above are numbered from 1 to 6 respectively.

Query number	Number of disk blocks used		
	2KB	4 KB	6 KB
1	9	5	4
2	9	5	4
3	259	130	87
4	259	130	87
5	259	130	87
6	740	371	247

Table 4.17: shows the query response time for the query set mentioned above

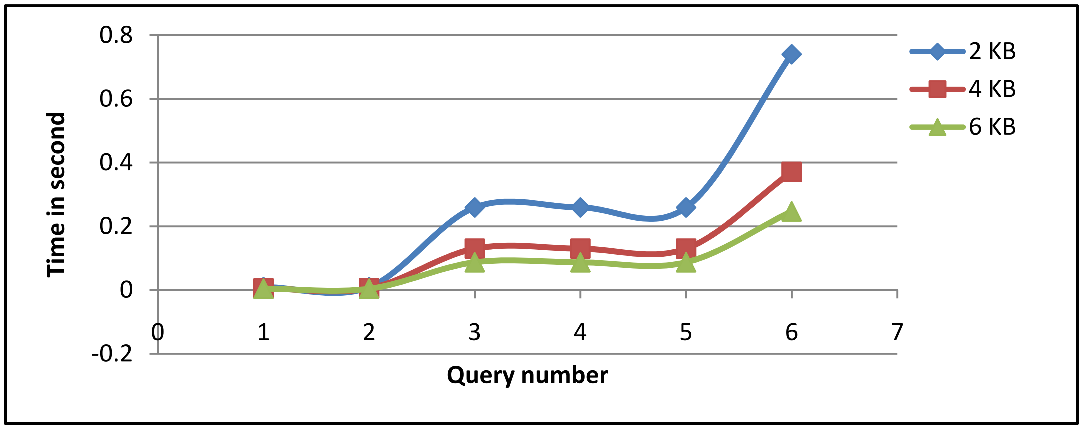


Figure 4.64: the query response time when querying SIGMODRECORD.xml datasets

4.5.2.2.3. Set C datasets

Three dataset groups will be implemented in this set:

4.5.2.2.3.1. Group C dataset

We have chosen "**ORDERS_GC.xml**" database; its size is 11.38 MB with 11 distinct elements and 9 distinct paths. The number of disk blocks when the size of the block is 2 KB is 1197, number of disk block when the size of the block is 4 KB is 603 and the number of disk blocks when the size of the block is 6 KB is 405 as shown in the following chart

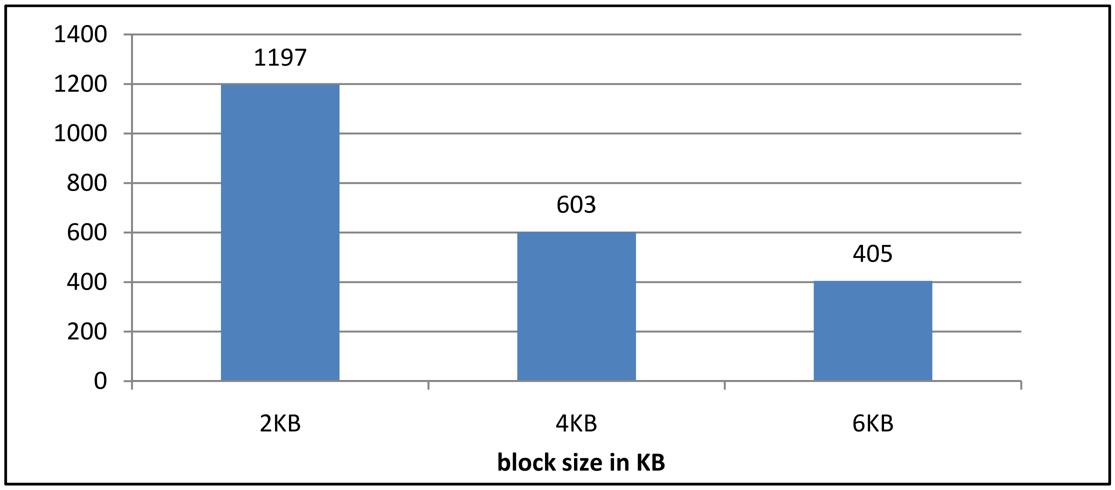


Figure 4.65: the number of blocks used to save ORDERS_GC.xml dataset

The density of the MXF file when the size of the block is 2KB is 99.12%, when the size of the block is 4KB, the density of the file is 98.38 %; and the file density when the size of the block is 6KB is 97.65% as shown in the chart bellow

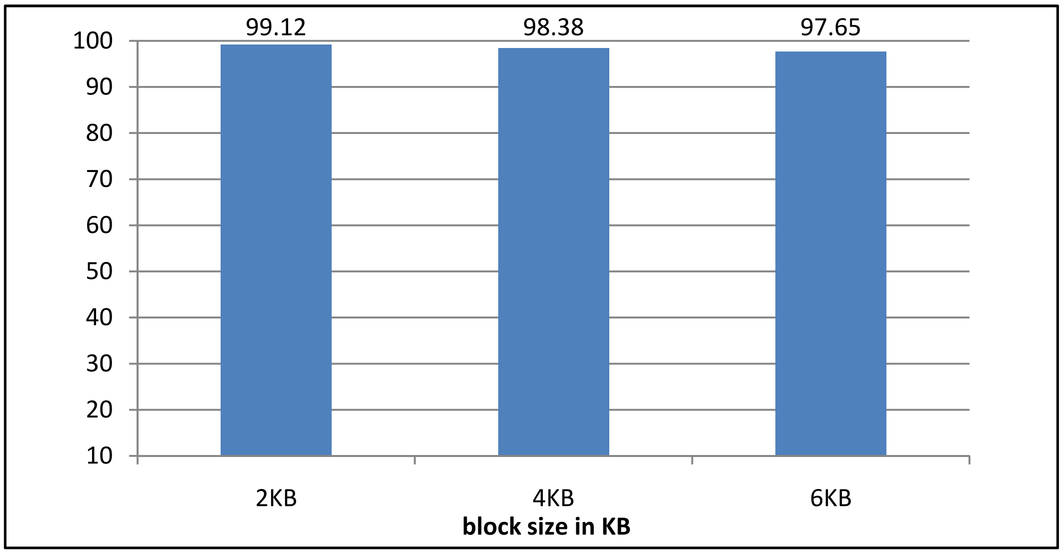


Figure 4.66: the file density when saving ORDERS _GC.xml dataset

4.5.2.2.3.2. Group B dataset

We have chosen "**ORDERS_GB.xml**" database; its size is 5.25 MB with 11 distinct elements and 9 distinct paths. The number of disk blocks when the size of the block is 2 KB is 603; the number of disk blocks when the size of the block is 4 KB is 306 and the number of disk blocks when the size of the block is 6 KB is 207 as shown in the following chart

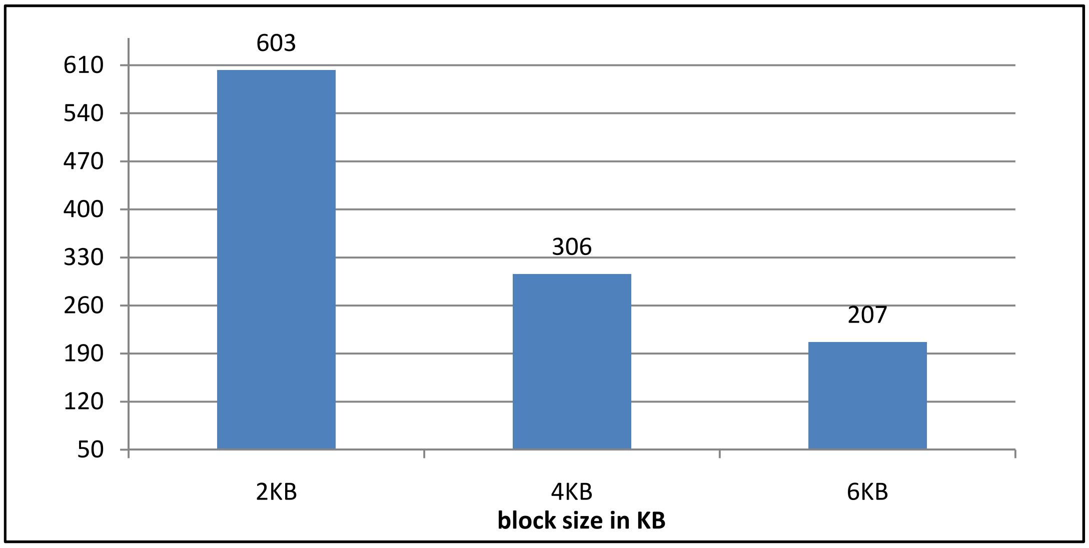


Figure 4.67: the number of blocks used to save ORDERS_GB.xml dataset

The density of the MXF file when the size of the block is 2KB is 98.38%, when the size of the block is 4KB, the density of the file is 96.93 %; and the file density when the size of the block is 6KB is 95.53% as shown in the chart bellow

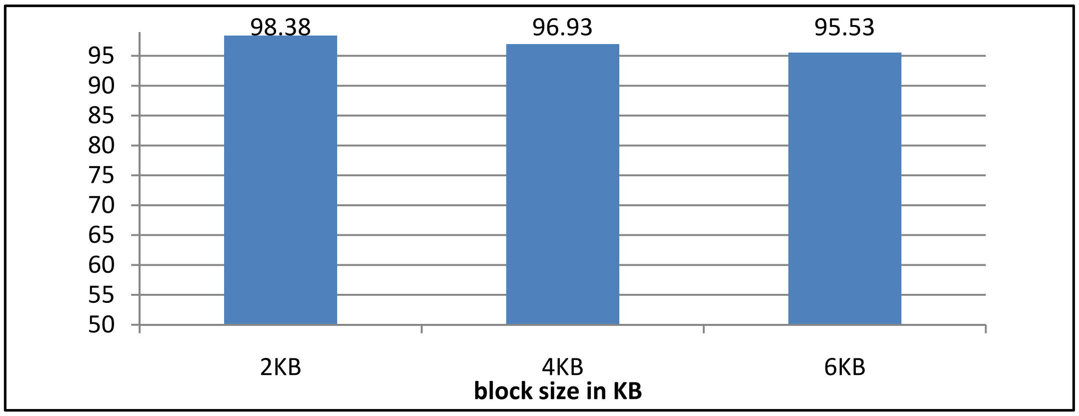


Figure 4.68: the file density when saving ORDERS _GB.xml dataset

4.5.2.2.3.3. Group A dataset

We have chosen "**ORDERS_GA.xml**" database; its size is 2.64 MB with 11 distinct elements and 6 distinct paths. The number of disk blocks when the size of the block is 2 KB is 279; the number of disk blocks when the size of the block is 4 KB is 144 and the number of disk blocks when the size of the block is 6 KB is 99 as shown in the following chart

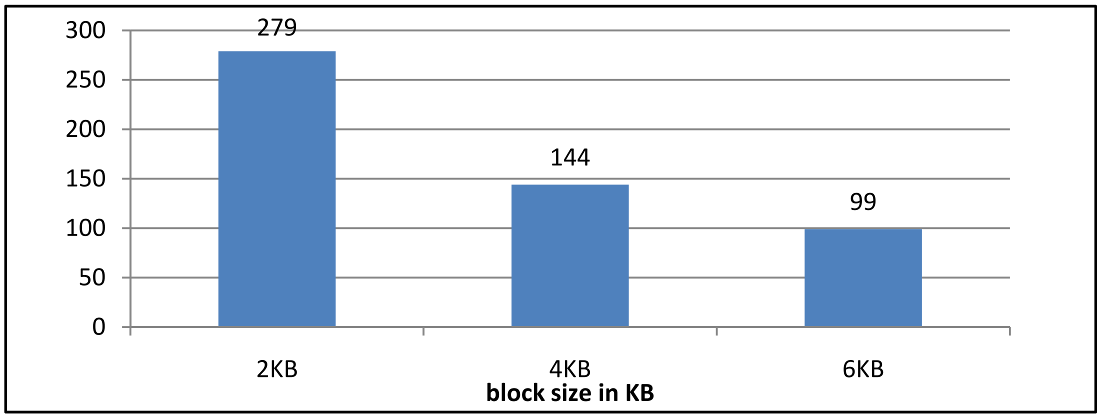


Figure 4.69: the number of blocks used to save ORDERS_GA.xml dataset

The density of the MXF file when the size of the block is 2KB is 96.66%, when the size of the block is 4KB, the density of the file is 93.64 %; and the file density when the size of the block is 6KB is 90.80% as shown in the chart bellow

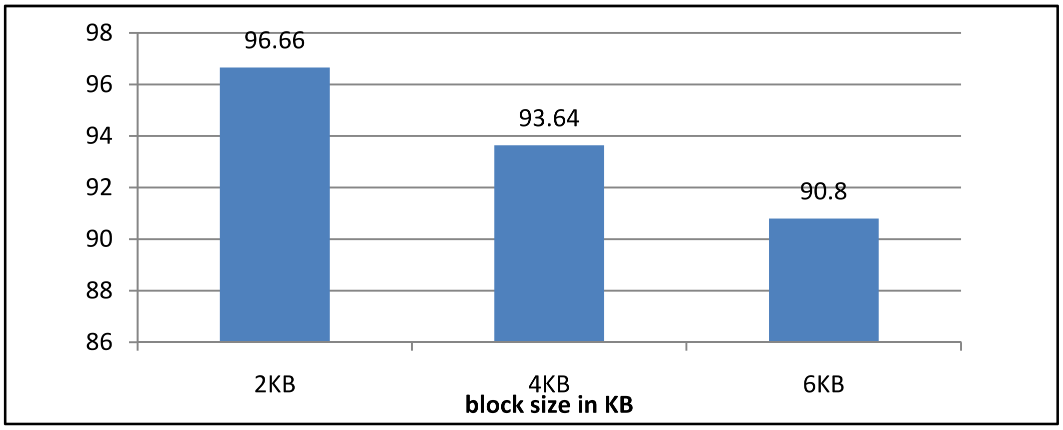


Figure 4.70: the file density when saving ORDERS _GA.xml dataset

Dataset	Size in MB	Number of disk blocks			File Density		
		2 KB	4 KB	6 KB	2 KB	4 KB	6 KB
ORDERS.xml	11.38	1197	603	405	99.67	99.16	99.08
	5.25	603	306	207	99.16	97.17	96.37
	1.51	64	32	24	96.37	93.13	88.45

Table 4.18: a summery table shows the density file and the number of blocks used when saving ORDERS.xml

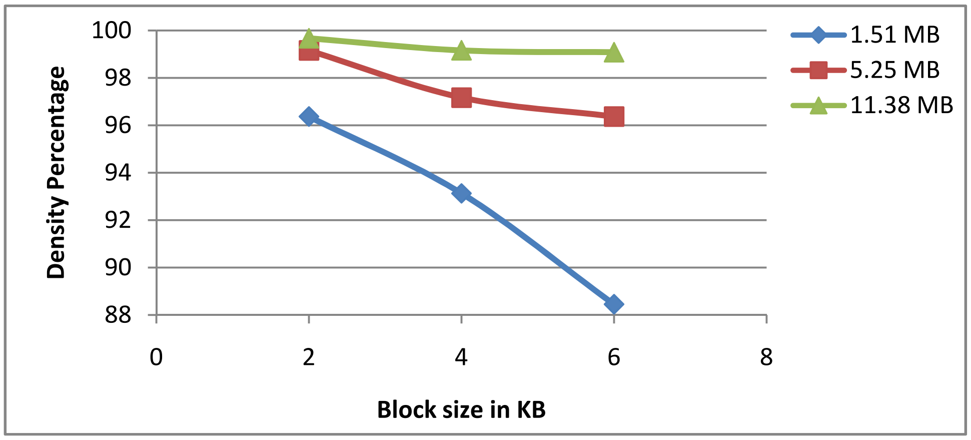


Figure 4.71 the density of MXF when saving ORDERS.xml datasets

"ORDERS.xml" has the smallest depth among "**SIGMODRECORD.xml**" and "**PERSONS.xml**", even if it has the highest density, but the density different is slight as shown in the following figure.

According to the response time of the queries applied on "**ORDERS.xml**" database, we have applied these simple path queries. These queries are T/O_ORDERKEY, T/O_CUSTKEY, T/O_ORDERSTATUS, T/O_TOTALPRICE, T/O_ORDERDATE, T/O_ORDER-PRIORITY, T/O_CLERK and T/O_SHIP-PRIORITY

Query number	**Number of disk blocks used**		
	2KB	**4 KB**	**6 KB**
1	16	8	6
2	40	20	14
3	19	10	7
4	14	8	5
5	129	65	44
6	15	8	6
7	21	11	8
8	21	11	3

Table 4.19: shows the query response time for the query set mentioned above

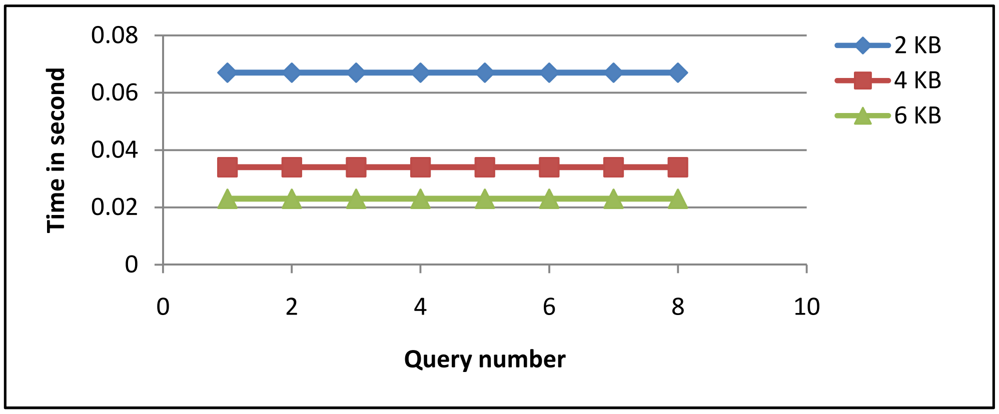


Figure 4.72: the query response time when querying WSU.xml datasets

4.6. Summery

In this chapter, we have implemented MXF on NINE datasets with different sizes and data. For each data set, we have shown the density of the file, the number of disk blocks used and the query response time for nine sets of simple path queries. Our datasets are classified according to two factors: the variety of the data and the shape of the data (width and depth). As the data becomes less distinct, the density of the file becomes higher; this in turn means fewer disk blocks will be used to save the data.

According to the width of the data**,** the width of the tree has a slight effect on the density of the file as well as on the number of the dick blocks used to save data. But as the size of the file becomes larger, it gives a chance for rising new distinct paths which can slightly affect the density of the file (in the rare case where the file can fit in very few disk blocks (like one or two disk blocks).

According to the depth factor, from the results we got above, we can conclude that: since we deal with BPL as a one unit, the depth of the file will not affect the file density as well as the number of disk blocks.

According to the query response time, the only factor that affects the query response time is the number of disk blocks that are used to save the corresponding data. The shape of the data will not affect response time as we don’t decompose the query to answer it; instead we take it as one unit i.e. the depth of the data is not a mater while querying XML data using MXF.

CHAPTER FIVE

CONCLUSION AND FEATURE WORK

Since the importance of XML as a new standard for information representation and exchange on the internet, the problem of storing, indexing, and querying XML documents poses new challenges to database researchers, and has been among the major issues of database research.

In this thesis, we proposed a new file structure called MXF to parse, index and store XML documents.

Previous native XML storage systems depend on the inverted lists to store the file. They save the inverted lists of all elements in an XML documents. They decompose the paths of the XML tree and save the elements of the paths separately.

There are two major disadvantages of these approaches: first, they cost large disk space since they store inverted lists of all elements, and the second is they need many join operations to process a query.

From these two major disadvantages, we contribute a new file structure Multidimensional XML File (MXF) to save, index and query XML document. The main idea of our proposed system is store the inverted lists of the leaf nodes only. This results in less disk space. Since we don’t decompose path into its elements, no need for join operations to answer a simple path query. MXF will minimize the cost of a query by minimizing the number of joins needed in case of twig queries.

We have applied MXF on nine datasets each with three different sizes (differ from each other in the data-variety and data-shape) and we conclude that our MXF is stable regardless the size, the shape and the data-shape of the XML data tree.

As a feature work, since we need to increase the file density in the situation mentioned above, we will develop a merge algorithm to merge more than one data block to increase the file density.

REFERENCES

[1] http://www.w3schools.com

[2] http://www.simonstl.com/articles/whyxml.htm

[3] http://www.w3.org/XML/

[4] Md. Sumon Shahriar and Jixue Liu, “On Defining Keys for XML”, IEEE 8th International Conference on Computer and Information Technology Workshops, 2008

[5] www.xml.com

[6] M. Altinel and J. Franklin, “Efficient filtering of XML documents for selective dissemination of information”, *VLDB Conference*, 2000.

[7] G. Gou, R. Chirkova, "Efficiently Querying Large XML Data Repositories: A Survey", IEEE Transactions on Knowledge and Data Engineering, Vol. 19, No. 10. (2007), pp. 1381-1403.

[8] G. Gou, R. Chirkova ,"XML Query Processing: A Survey",2005

[9] Rebecca J. Cathey, Steven M. Beitzel, Eric C. Jensen, D Grossman, O Frieder, "Using a Relational Database for Scalable XML Search", 2007

[10] J. Leonard," Strategies for Encoding XML Documents in Relational Databases: Comparisons and Contrasts", Thesis 2006

[11] Mustafa Atay , Yezhou Sun , Dapeng Liu , Shiyong Lu ad Farshad Fotouhi, “Mapping Xml Data To Relational Data: A Dom-Based Approach” (2004),Eighth IASTED International Conference on Internet and Multimedia Systems and Applications, Kauai

[12] S. Chaudhuri, K. Shim," Storage and Retrieval of XML Data using Relational Databases", Tutorial. ICDE 2003.

[13] S. Prakash, S. B. Bhowmick, S. Madria." Efficient Recursive XML Query Processing Using Relational Database Systems", In Proceedings of ER. 2004

[14] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt and J.F. Naughton, "Relational Databases for Querying XMLDocuments: Limitations and Opportunities," Proc. 25th Int'l Conf.Very Large Data Bases (VLDB '99), 1999.

[15] G. Pavlovic," Native XML Databases vs. Relational Databases in dealing with XML Documents", 2007

[16] H. V. Jagadish et al, "A Native XML Database", In International Conference of VLDB. 2002.

[17] http://www.xml.com/pub/a/2001/10/31/nativexmldb.html

[18] A Vakali, B Catania and A Maddalena, "XML Data Stores: Emerging Practices", Published by the IEEE Computer Society 1089-7801/05/$20.00 © 2005 IEEE

[19] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: A Database Management System for Semistructured Data," SIGMOD Record, vol. 26, no. 3, pp. 54-66, 1997.

[20] J. McHugh and J. Widom, "Query Optimization for XML," Proc.25th Int'l Conf. Very Large Data Bases (VLDB '99), 1999.

[21] P.F. Dietz, "Maintaining Order in a Linked List," Proc. 14th ACM Symp. Theory of Computing, 1982.

[22] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G.M. Lohman,"On Supporting Containment Queries in Relational DatabaseManagement Systems," Proc. 20th ACM SIGMOD Int'l Conf.Management of Data (SIGMOD '01), 2001.

[23] Online Computer Library Center, "Dewey decimal classification", http://www.oclc.org/dewey/, 2006.

[24] I. Tatarinov, S. Viglas, K.S. Beyer, J. Shanmugasundaram, E.J. Shekita, and C. Zhang, "Storing and Querying Ordered XML Using a Relational Database System," Proc. 21st ACM SIGMODInt'l Conf. Management of Data (SIGMOD '02), 2002.

[25] Cohen, E., Kaplan, H., Milo, T," Labeling dynamic XML trees. In: Proceedings of PODS 2002, pp. 271–281.

[26] X. Wu, M. Li Lee, W. Hsu," A Prime Number Labeling Scheme for Dynamic Ordered XML Trees", 2004

[27] M. Altinel and J. Franklin, "Efficient filtering of XML documents for selective dissemination of information", *VLDB Conference*, 2000.

[28] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation- vs. index-based XML multi-query processing. *ICDE Conference*, 2003.

[29] Yanlei Diao and Michael J. Franklin," High-performance XML filtering: an overview of YFilter", *IEEE Data Engineering Bulletin*, 26:41–48, 2003.

[30] D. Florescu and D. Kossmann, "Storing and Querying XML Data Using an RDMBS," IEEE Data Eng. Bull., vol. 22, pp. 27-34, 1999.

[31] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, “XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases,” ACM Trans. Internet Technology, vol. 1, pp. 110-141, 2001.

[32] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V.Zolotov, “Indexing XML Data Stored in a Relational Database,”Proc. 30th Int’l Conf. Very Large Data Bases (VLDB ’04), 2004.

[33] http://www.dataexmachina.de/natix.html

[34] http://www.eecs.umich.edu/db/timber

[35] www.ipedo.com

[36] http://xml.apache

[37] http://exist.sourceforge

[38] www.dbXML.com

[39] *Q. Li and B. Moon,”* Indexing and Querying XML Data for Regular Path Expressions” , Proceedings of the 27th International Conference on Very Large Databases (VLDB'2001), pages 361-370, Rome, Italy, September 2001.

[40] N. Zhang, V. Kacholia, and M.T. Ozsu, “A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML,” Proc. 20th IEEE Int’l Conf. Data Eng. (ICDE ’04), 2004.

[41] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava,and Y. Wu,“Structural Joins: A Primitive for Efficient XML QueryPattern Matching,” Proc. 18th IEEE Int’l Conf. Data Eng. (ICDE ’02), 2002

[42] N. Bruno, N. Koudas, and D. Srivastava, “Holistic Twig Joins:Optimal XML Pattern Matching,” Proc. 21st ACM SIGMOD Int’lConf. Management of Data (SIGMOD ’02), 2002.

[43] V. GAEDE, O. GU¨NTHER, ” Multidimensional Access Methods”, ACM Computing Surveys, Vol. 30, No. 2, June 1998

[44] J. NIEVERGELT, H. HINTERBERGER,”The Grid File: An Adaptable, Symmetric

Multikey File Structure”, ACM Transactions on Database Systems, Vol. 9, No. 1, March 1984

Vita

- Mahboub Ali Mohammed Naji.
- Nationality: Yemeni
- Born in Yemen on September 1, 1979.
- Completed Bachelor of Science (B.Sc.) in Computer Science from Thamar University, Yemen, in June 2001.
- Present address: King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia
- Permanent address: Yemen, Taiz; Email: te_mahboob@yahoo.com