Department of Informatics Faculty of
Mathematics, Physics and Informatics
Comenius University, Bratislava

Optimization of the Native XML Database Deployed at the blog.matfyz.sk Portal

(Master thesis)

Bc. Eva Lichnerová

Thesis advisor: RNDr. Martin Homola, PhD.

Bratislava, 2012

Optimization of the Native XML Database Deployed at the blog.matfyz.sk Portal

Master thesis

Bc. Eva Lichnerová

COMENIUS UNIVERSITY, BRATISLAVA, SLOVAKIA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF INFORMATICS

2508 Informatics

Thesis advisor: RNDr. Martin Homola, PhD.

BRATISLAVA 2012

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Eva Lichnerová
Study programme:	Computer Science (Conversion Programme) (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Optimization of the Native XML Database Deployed at the blog.matfyz.sk Portal

Aim: To analyze the current installation of the Sedna native XML DBMS used by the blog.matfyz.sk portal and to propose and implement performance fixes with the aim to reduce crashes and to boost the query response time.

Literature: Kohutovic, A. (2008). blog.matfyz.sk community portal. Master’s thesis, Comenius University, Faculty of Mathematics Physics and Informatics, Bratislava, Slovakia.

Rejda. M. (2010) Modular redesign of the blog.matfyz.sk portal. Master’s thesis, Comenius University, Faculty of Mathematics Physics and Informatics, Bratislava, Slovakia.

Sedna Documentation: http://modis.ispras.ru/sedna/documentation.html

Annotation: The current database system used at the blog.matfyz.sk portal suffers from occasional crashes and slow response times at heavy loads. This is due to the DBMS was deployed by other students whose major tasks was different. The student's job is threefold: 1) To optimize the current installation and configuration of the system. 2) To analyze and optimize the queries executed from the portal code 3) If this does not help - to analyze the code of the Sedna DBMS and seek for sub-optimal passages, especially aiming features such as indexing, caching, etc. to propose the solution and try to fix it.

As the last resort, student may choose to replace the DBMS by some other that satisfies the needs of the portal.

Sedna, a state-of-the art native XML DBMS is developed by the MODIS group at the Russian Academy of Sciences. It is an open source product. The student is expected to communicate with the Sedna development team, especially working towards the goal no. 3. The collaboration was established in the past.

Supervisor: RNDr. Martin Homola, PhD.
Department: FMFI.KI - Department of Computer Science

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Assigned: 17.10.2010

Approved: 03.05.2012 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

....

Student Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Eva Lichnerová
Študijný program:	informatika (konverzný program) (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Optimalizácia natívnej XML databázy využívanej na portáli blog.matfyz.sk

Cieľ: Analyzovať súčasnú inštaláciu natívnej XML databázy Sedny používanej na portáli blog.matfyz.sk a navrhnúť a implementovať výkonnostné úpravy s cieľom redukovať výpadky a zrýchliť čas vykonávania príkazov.

Literatúra:

Kohutovic, A. (2008). blog.matfyz.sk community portal. Master's thesis, Comenius University, Faculty of Mathematics Physics and Informatics, Bratislava, Slovakia.

Rejda. M. (2010) Modular redesign of the blog.matfyz.sk portal. Master's thesis, Comenius University, Faculty of Mathematics Physics and Informatics, Bratislava, Slovakia.

Sedna Documentation: http://modis.ispras.ru/sedna/documentation.html

Anotácia: Súčasný databázový systém používaný na portáli blog.matfyz.sk trpí občasnými výpadkami a dlhými časmi vykonávania príkazov pri veľkej záťaži. Príčinou je, že databázový systém bol nasadzovaný inými študentmi, ktorých hlavná úloha bola odlišná od optimalizácie. Práca študenta je rozdelená na tri časti: 1) Optimalizovať súčasnú inštaláciu a konfiguráciu systému. 2) Analyzovať a optimalizovať príkazy vykonávané z kódu portálu. 3) Ak toto nepomôže - analyzovať kód Sedny a vyhľadať menej optimálne časti, konkrétne sa zamerať na prvky ako indexovanie, cachovanie, atď., navrhnúť riešenie a pokúsiť sa o opravu.

Poslednou možnosťou je, že študent si vyberie iný databázový systém, ktorý viac vyhovuje požiadavkám portálu a povôdný systém ním nahradí.

Sedna, jedna z najnovších natívnych XML databáz, je vyvíjaná skupinou MODIS pôsobiacou v Ruskej akadémii vied. Je to open source produkt. Od študenta sa očakáva komunikácia s vývojovým tímom Sedny, špeciálne pri plnení cieľa č.3. Spolupráca bola nadviazaná už v minulosti.

Vedúci: RNDr. Martin Homola, PhD.
Katedra: FMFI.KI - Katedra informatiky

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátum zadania: 17.10.2010

Dátum schválenia: 03.05.2012

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....
študent

....
vedúci práce

I would like to thank my supervisor Martin Homola for his excellent guidance, support and advices. The next thanks goes to my ex-colleague Martin Rejda, who introduced me into the secrets of XML databases, and Ivan Shcheklein from the Institute for System Programming of the Russian Academy of Sciences, a member of the Sedna development team, who answered patiently all of my questions. The special thanks goes to my family for their big support, not only during writing this thesis, but throughout my whole study. Thank you very much.

Abstrakt

XML formát je v dnešnej dobe veľmi využívaný, čo vedie aj k narastajúcemu používaniu XML databáz. Keďže tieto databázy sú ešte relatívne novým prírastkom v skupine databázových systémov, sú stále intenzívne vyvíjané a zdokonaľované. XML databáza nasadená na portáli blog.matfyz.sk nebola nikdy optimalizovaná s úmyslom dosiahnuť jej najlepší výkon. To malo za následok časté výpadky databázy a veľmi časté boli aj príkazy, ktorých vykonávanie trvalo neprimerane dlho. Cieľom tejto práce bolo optimalizovať XML databázu nasadenú na našom portáli. V práci sme predstavili pozadie XML databáz a detailne sme ich popísali. Ďalšou časťou práce bolo analyzovať existujúce optimalizácie systému a navrhnúť nové optimalizačné techniky, ktoré by mohli byť použité počas vykonávania merania. Vytvorili sme dva typy meraní, ktoré merali, ako sa podarilo zrýchliť čas vykonávania samostatného príkazu a celej akcie (napr. načítanie domovskej stránky portálu) pomocou navrhnutých optimalizačných techník. Výsledky meraní ukázali, že tieto optimalizačné techniky výrazne zvýšili výkon našej XML databázy.

Kľúčové slová: XML, Natívna XML databáza, Optimalizácia

Abstract

The XML format is widely used nowadays what leads to incremental usage of XML databases too. Since they are relatively new in the database management systems group, they are still intensively developed and improved. The XML database deployed at the blog.matfyz.sk portal was never optimized to gain its maximal performance. This led to database crashes and long execution time of statements. The main goal of this work was to optimize the XML database deployed at our portal. In this work we introduced the background of the XML databases and described them in detail. Next part of the work was to analyse the existing database optimization and to propose optimization techniques, which could be used during the benchmarking. We created two benchmarks, which measured the improvement of the statement and action (e.g., homepage loading) execution time using proposed optimization techniques. The benchmark results showed that these optimization techniques boosted the database performance significantly.

Keywords: XML, Native XML database, Optimization

Preface

This work was created with purpose to propose the best optimization solution for the XML databse deployed at the blog.matfyz.sk portal. Since the portal was developed by many students, who had their assignments different from the database optimization, this led to state, where the database was crashing and statements were very time consuming. This created the need to optimize current database installation properly. This work describes, which techniques should be used for optimization and how these techniques improved our database performance. This work is intended for everyone, who is interested in XML, XML databases or database optimization.

Contents

Introduction **1**

1 XML **3**

1.1 Background . . . 3
1.2 XML data structure . . . 4
1.2.1 Structural components . . . 5
1.2.2 Content type categories . . . 10
1.3 XML document type declaration . . . 10
1.3.1 Element type declaration . . . 12
1.3.2 Attribute-list declaration . . . 12
1.4 XML querying and transformation . . . 12
1.4.1 DOM . . . 13
1.4.2 XPath . . . 13
1.4.3 XSLT . . . 14
1.4.4 XQuery . . . 16
1.4.5 XQuery Update facility . . . 19

2 XML databases **21**

2.1 Background . . . 21
2.2 Native XML database . . . 23
2.2.1 Storage architecture . . . 23
2.2.2 Main features . . . 24

3 Sedna **29**

3.1 Architecture 29

3.2 Storage system 31

3.2.1 Data organization 31

3.2.2 Memory management 32

3.2.3 Index management 34

3.3 Update language 35

3.4 Transactions 37

3.4.1 Multiversioning 37

3.4.2 Locking 38

3.4.3 Read-only transactions 38

3.4.4 Recovery 38

4 Optimization **39**

4.1 Why the database needs an optimization 39

4.2 Hardware and software analysis 40

4.2.1 Server analysis 40

4.2.2 Sedna analysis 40

4.2.3 Our database structure 42

4.3 Existing optimization 44

4.4 How to optimize 44

4.4.1 Sedna settings 44

4.4.2 PHP API function options 45

4.4.3 Statements optimization 46

4.5 Optimization results measurement 47

5 Benchmarks **49**

5.1 Existing XML benchmarks 49

5.2 Choosing an appropriate benchmark 51

6 Statement analysis **53**
6.1 Gathering the data 53
6.2 The data analysis 54
6.2.1 Overall analysis 54
6.2.2 Different statements usage 56
6.2.3 Predicates usage 58

7 Portal benchmarks **63**
7.1 Our benchmark design overview 63
7.2 Simple benchmark 64
7.2.1 Specification 64
7.2.2 The results 68
7.3 Complex benchmark 71
7.3.1 Specification 71
7.3.2 The results 77

Conclusion **83**

A Simple benchmark **91**
A.1 Operation sets 91
A.1.1 Original statements set 91
A.1.2 Optimized statements set 96
A.1.3 Index statements set 99
A.1.4 Index II statements set 103
A.2 Partial results 104

B CD content **107**

Introduction

Background

Blogs are social phenomenon today. Everybody interested in the internet knows what a word “blogger” means. Our blog.matfyz.sk portal was created in 2008 as a project with two purposes. The first purpose was to develop a portal where technologies as XML databases, XSLT, ranking algorithms could be researched in the real world. The second purpose was to develop a portal where students, teachers and their friends could meet and discuss not only about the school.

Nowadays the portal is used in a learning process also – there is a bunch of courses, which use the portal for assigning homeworks, midterms, exams and for student evaluation. The portal is developed mainly by the students who have their bachelor and master thesis associated with the portal. This development is still in progress and new feautures are planned to deploy – a sentiment analysis of posts and comments, real-time xml editor, etc.

Motivation

Since the portal was developed by many students, whose primar goal was different from database statements and settings optimization, there where significant performance problems with the XML database deployed at the portal. Long execution times of statements and crashes of the database led to necessity to optimize the database statements and settings and try to gain the maximal performance from the database.

Problem

We needed to analyse the database settings, their usage, what was done for the database optimization in the past and how we could optimize the database statements and settings. The next step was to choose an appropriate tool for the optimization results measurement. There exists benchmarks for XML databases, but as it showed, these were not appropriate for our purpose, so we decided to create our own benchmark tests for the optimization results measurement.

Contribution

The main contribution of this work is that we boosted the database performance significantly comparing to original database performance. We achieved that each basic action executed on the portal (e.g., homepage loading, an insertion of the post or a comment) was boosted considerably. This database performance optimization caused that currently the users' experience of the portal is incomparably more positive than before the optimization – the portal pages are loaded smoothly and fast, there are no database crashes and users do not have a problem with the portal usability.

Chapter 1

XML

1.1 Background

Extensible Markup Language (XML) is well-known in a lot of areas nowadays. This language was created in 1996 and was derived (as *Hyper-Text Markup Language* - HTML) from *Standard Generalized Markup Language* (SGML), which was designed for structuring large documents. The XML main design goals were: it had to be straightforwardly usable over the internet, support wide variety of applications, it had to be human-legible and reasonably clear, formal and concise and easy to create [W3C08]. XML was designed to carry data, not to display them and thanks to this attribute it is a perfect language for storing data as well.

The term *markup* refers to anything in a document that is not intended to be a part of a printed output [SKS01]. So a markup language is a formal description of what a markup is, what in document is content and what that markup means in a context. Markup languages evolved from specifying instructions how to print content to specifying the function of that content and this is their main contribution. A *tag* defined by *tagname* enclosed in angle-brackets `<,>` is the markup form. Tags are commonly used in pairs as a *start-tag* and *end-tag* – `<tag> </tag>`. The first delimits the beginning and the second delimits the end of content, to which this tag refers. There is a short example in Example 1.1.

Example 1.1 Tag usage

```
<name>Eva Lichnerova</name>
```

Tags in XML are not prescribed as they are in HTML – in XML we can create any tags we want and need. This feature makes XML very powerful tool for a data representation and their exchange. Other advantages of this language are:

- XML is self-descriptive – we do not need anything else to understand the meaning of the text.
- We can add our own tags to existing document and our recipient will still be capable of reading the whole document. This means that documents can evolve without invalidating existing applications, which depend on them.

Currently, the XML format is widely accepted and there are many tools, which can read and process any XML document, what makes XML the most widely-used format of its kind.

1.2 XML data structure

Definition 1.2.1 ([W3C08]) *Markup takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, processing instructions, XML declarations, text declarations, and any white space that is at the top level of the document entity (that is, outside the document element and not inside any other markup).*

Each XML document has its physical and logical structure. The physical structure is created by units called *entities*. Entities have content and are identified by entity names. The document logically consists of declarations, elements, comments, character references and processing instructions, all of which are indicated in the document by explicit markup [W3C08]. In the following section we describe some of the structural components in detail.

1.2.1 Structural components

Elements

Definition 1.2.2 ([W3C08]) *Each XML document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications.*

Example 1.2 Short example of an XML document

```
<?xml version="1.0"?>
<blog>
  <title>My new blog</title>
  <post status="published" ID="p81">
     <title>Blogging</title>
     <content>Test content</content>
     <comments/>
  </post>
</blog>
```

The majority of an XML document is created by elements. Each XML document has one top-level element, called *the root* or *document element*, which is not a part of any other element. As Definition 1.2.2 says, an element is defined by a start-tag and end-tag, with its content between these tags (see Listing 1.1). If an element is empty (it has no content), we can mark it as `<tag/>`, what is a shorthand for `<tag></tag>`. An element name is case-sensitive and must begin with a letter or an underscore (_). Content of an element is created by its children. These children can be in form of other elements, comments, CDATA sections or characters and are ordered. Each element may be annotated with attributes also, but unlike children, they are unordered. Example 1.2 contains a short XML document with mentioned characteristics.

Attributes

Attributes are used to annotate elements – they provide extra information about them. Attributes are placed inside of a start-tag and appear as name=value pairs separated by equal sign, see Example 1.2. An attribute value is textual and may be enclosed by single or double quotes. The number of attributes in one element is not restricted, but attributes must have different names.

Namespace declarations

Example 1.3 XML document using namespaces

```
<?xml version="1.0"?>
<blog xmlns:portal="http://blog.matfyz.sk/portal/"
    xmlns="http://blog.matfyz.sk/example/">
  <title>My new blog</title>
  <portal:post status="published" portal:ID="p81">
    <portal:title>Blogging</portal:title>
    <portal:content>Test content</portal:content>
    <comments/>
  </portal:post>
</blog>
```

A single XML document may contain elements and attributes (referred as *markup vocabulary*) that are defined for and used by, e.g., multiple software modules. Such documents, containing multiple markup vocabularies, may cause recognition and collision problems. Software modules need to be able to recognize the elements and attributes which they are designed to process, even if the collision occurs when markup intended for some other software package uses the same element name or attribute name. These considerations require that document constructs should have names constructed uniquely between different markup vocabularies. This goal is achieved by *XML Namespaces*, which assign extended names to elements and attributes [W3C09].

Each namespace has a name – a *uniform resource identifier* (URI), which serves as a unique string. The namespace name and the name of an element/attribute create globally unique name called *qualified name*. Namespace

declarations appear in this form – `xmlns:prefix="URI"` (single quotes may be used too) [W3C09] – and are placed in an element start-tags. Namespace declaration map name of a namespace to another (usually shorter) string, *namespace prefix*. An element can contain any number of declarations of this type (called nondefault), but they must have different prefixes. Default namespace declarations appear as `xmlns="URI"` – without prefix defined. An element cannot contain more than one default namespace declaration.

A scope of the namespace declaration is an element, in which the declaration is placed, and all its children. Certain namespace prefix can be used only in its declaration scope elements/attributes, otherwise an error is raised. Extended element/attribute name syntax is `prefix:name`. The namespace of unprefixed element/attribute names is the namespace specified by the in-scope default namespace declaration, if any. If there is no in-scope default namespace declaration, such elements/attributes are called *unqualified* [SG01]. An illustration of the XML document using namespaces is in Example 1.3.

Comments

Comments are used to provide additional information to humans about XML document content. They may appear anywhere in the document, but outside other markup. A comment is bounded by these character sequences `<!-- -->` with comment content placed between them. For compatibility reasons the comment content must not contain double-hyphen (`--`) [W3C08]. Comment usage is in Example 1.4.

Processing instructions

Processing instructions are used to provide additional information to applications, which process the XML document. Instructions may contain information, e.g., how to display content or how to process the document. The syntax of processing instruction takes the form `<?target data?>`, where `target` is the name of a processing instruction and `data` are information for this instruction. Processing instructions can appear as children of elements or as top-level constructs before the document element.

Example 1.4 XML document using comments, CDATA section, XML declaration, entity references

```
<?xml version="1.0" encoding="UTF-8"?>
<blog>
  <!-- Title of the blog -->
  <title>My new blog</title>
  <post status="published" ID="p81">
    <title>Blogging &amp; chatting</title>
    <content>
      <![CDATA[Test content & some other stuff.]]>
    </content>
    <comments/>
  </post>
</blog>
```

CDATA sections

Definition 1.2.3 ([W3C08]) *CDATA sections may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "* <![CDATA[*" and end with the string "*]]> *".*

CDATA sections can appear inside element content and allow usage of such characters as <,> or & (see Example 1.4), which would be otherwise processed as markup characters. The only markup recognized by an XML processor is a character sequence, which ends a CDATA section –]]>. CDATA sections cannot be nested [SG01].

XML declaration

Definition 1.2.4 ([W3C08]) *XML documents should begin with an XML declaration which specifies the version of XML being used.*

As Definition 1.2.4 says, an XML declaration does not have to be used in the XML document. If it is presented, it has to be the first construct in the document. The XML declaration begins with the character sequence <?xml and ends with ?> (see Example 1.4). Although its syntax is the same as the

syntax of processing instructions, the XML declaration is not considered as a processing instruction. Declaration content consists of three attributes – `version`, `encoding` and `standalone`. Only `version` attribute is mandatory, others are optional. Attribute order has to be retained in this form [SG01].

Entity references

Certain characters may cause problems, if they are used in content of the element or as attribute values. Such character is, e.g., less-than `<`, which represents the beginning of the start-tag or end-tag. If we want to write such character, we have to use an alternative way to represent it. Entities are used to represent these special characters in content. Since their syntax has a form `&entity_name;`, their names must be unique [W3C08]. For less-than character there is an entity `<`, for ampersand `&` (see Example 1.4).

Character references are special case of entity references. They are used for inserting an arbitrary character into the document. The syntax form is the same as entity reference form, but `entity_name` is replaced by decimal or hexadecimal reference of the character in Unicode.

Well-formed XML document

A textual object is a well-formed XML document, if it satisfies these conditions [SG01]:

- all constructs must be syntactically correct
- there must be exactly one top-level element
- each start-tag must have its end-tag
- all tags must be nested properly

Valid XML document

Definition 1.2.5 ([W3C08]) *An XML document is valid if it has an associated document type declaration and if the document complies with the constraints expressed in it.*

What is the document type declaration and how the constraints look like we explain in detail in the next section.

1.2.2 Content type categories

XML documents can be divided into two categories based upon their content [Bou05]:

Document-centric Document-centric documents are usually documents that are designed for human consumption. Examples are books, emails, advertisements, etc. They commonly have an irregular structure, larger grained data and lots of mixed content. The order, in which elements occur, is almost always significant.

Data-centric Data-centric documents are primarily designed for a computer program, which processes the information, responds to it, stores data items in a database, and so on. They have a regular structure, fine-grained data and they do not contain any mixed content usually. Examples of data-centric document are flight schedules or sales orders.

1.3 XML document type declaration

One of the main features of XML is, we can define our own tagnames and easily create an arbitrary XML document. But applications, which work with XML documents, need to know, how the processed XML documents should look like, so they can determine a correct or incorrect XML document. There have to be some constraints, which define used tags, their sequence or nesting.

Definition 1.3.1 ([W3C08]) *The XML document type declaration contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD. The document type declaration can point to an external subset (a special kind of external entity) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.*

Definition 1.3.2 ([W3C08]) *A markup declaration is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration.*

The XML document type declaration is an optional part of the XML document. If it is present, it must appear before the first element in the document [W3C08]. The document type declaration identifies the top-level element of the document and may contain additional declarations. The additional declarations may come from an external DTD, called the *external DTD subset*, or be included directly in the document, the *internal DTD subset*, or both (see Example 1.5).

In next sections we briefly describe first two markup declarations mentioned in Definition 1.3.2.

Example 1.5 The document type declaration usage in the XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE blog SYSTEM "blog.dtd"[
  <!ELEMENT blog (title, post*)>
  <!ELEMENT post (title, content, comments)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT content ANY>
  <!ELEMENT comments (comment*)>
  <!ATTLIST post
    ID ID #REQUIRED
    status CDATA #IMPLIED>
]>
<blog>
  <title>My new blog</title>
  <post status="published" ID="p81">
    <title>Blogging</title>
    <content>Post content.</content>
    <comments/>
  </post>
</blog>
```

1.3.1 Element type declaration

Defines an element of the specified name with the specified content model. The content model specifies what children can be used as element content, their order and number of occurrences. The content model can contain keywords `ANY` (any child is allowed), `EMPTY` (no children are allowed) or a child group definition enclosed in brackets (`#PCDATA` indicates that only text is allowed). See detailed Example 1.5.

1.3.2 Attribute-list declaration

Defines a set of attributes, which is allowed in some element. Each attribute has a name, type and default declaration (see Example 1.5). We distinguish these types of attributes – `CDATA` (character data), `ID` (a unique name), `IDREF` (a reference to an `ID`), `IDREFS` (a list of `IDREF` values), `ENTITY` (unparsed entity name), `ENTITIES` (a list of `ENTITY` values), `NMTOKEN` (a valid XML name), `NMTOKENS` (a list of `NMTOKEN` values). A default declaration can be specified as a simple string value, `#REQUIRED` (an attribute value must be always specified), `#IMPLIED` (an attribute is optional) or `#FIXED "value"` (an attribute is optional, but if it is present, it must have defined value).

1.4 XML querying and transformation

Tools, which provide an effective management of XML data, are becoming increasingly important nowadays because of a huge amount of applications using an XML format for storing and exchange data. These tools, created for an XML querying and transformation, are essential, when we talk about extracting information from an XML document and converting data between different XML representations.

Several languages provide different degree of querying and transformation capabilities of an XML document: *DOM*, *XPath*, *XSLT*, *XQuery* and *XQuery Update facility*. All these languages use a tree model of XML data during processing. Each XML document is logically modelled as a tree – it has nodes

and every node corresponds to an element or an attribute (see Section 1.2) in that XML. Nodes can have children nodes and every node except a root node (represents a root element) has a parent node. This structure creates a hierarchy, in which every node has a specific position, so we can distinguish a parent, a child, an ancestor, a descendant or a sibling node.

In next sections we describe each of these languages in more detail.

1.4.1 DOM

The *Document Object Model* is a platform- and language-neutral interface that allow programs and scripts to dynamically access and update the content, structure and style of XML documents [W3C05]. The XML DOM views an XML document as a tree structure and defines a standard way for accessing and manipulating XML documents. All elements can be accessed through the DOM tree. Their content (text and attributes) can be modified or deleted, and new elements can be created. The elements, their text, and their attributes are all known as nodes.

1.4.2 XPath

The *XML Path Language* is designed to allow a selection of a specific part or parts of the XML document for next processing. The basic building block of XPath is the expression, which is a string of Unicode characters. XPath defines a tree model of the XML document against which all expressions are evaluated - the *XPath data model*. There are seven kinds of nodes in the data model: document, element, attribute, text, namespace, processing instruction, and comment [W3C10d].

Most of XPath expressions are used for identifying the set of nodes within the trees. This type of expressions is called a *path expression* (see Simple path expression in Example 1.6). It looks like a file system path, except it navigates through the XPath tree model. A path expression consists of series of one or more steps, separated by `/` or `//`, and optionally beginning with `/` or `//`. A step generates a sequence of items and then filters the sequence by zero or more predicates (predicates are expressions enclosed in square brackets). The

value of the step consists of those items that satisfy the predicates, working from left to right. A step may be either a node test or a primary expression. A node test returns a sequence of nodes that satisfies a specified node name or kind (element, attribute, etc.) condition. Primary expression is , e.g., a literal, a variable reference, a parenthesized expression, a function call or a context item expression [W3C10b]. See Example 1.6 for more details.

Other XPath expressions are *sequence expressions, arithmetic expressions, comparison expressions, logical expressions, for expressions, conditional expressions* and *quantified expressions*. Each of them is described in detail in [W3C10b].

Function and operation library for XPath is specified in a separate document [W3C10e] and is supported also by XQuery 1.0 and XSLT 2.0. This library is very complex. It contains, e.g., these numeric, string, boolean, date and time, node or sequence functions:

- numeric: abs(), floor(), round()
- string: compare(), concat(), contains()
- boolean: true(), false(), not()
- date and time: hours-from-duration(), minutes-from-dateTime(), seconds-from-time()
- node: name(), number(), root()
- sequence: empty(), count(), id()

1.4.3 XSLT

The *XML Stylesheet Language Transformations* is a declarative programming language, written in XML, for converting XML documents to other text formats [W3C07]. These output formats are usually HTML or XML, but in principle, XSLT can convert XML to any text format. Converting XML to XML is useful mainly in those situations, when we need XML to conform to declaration different from original. Since version 2.0, the input format do not have to be only XML, but also other input formats can be used, e.g., CSV.

XSLT uses XPath for a selection of XML parts and works with the same

Example 1.6 XPath – path expressions

```
Queried XML document:

  <?xml version="1.0"?>
  <blog>
    <title>My new blog</title>
    <post accessCount="123" ID="p81">
       <title>Blogging</title>
       <content>Test content</content>
       <comments/>
    </post>
    <post accessCount="234" ID="p94">
       <title>W3C Standards</title>
       <content>Content</content>
       <comments/>
    </post>
  </blog>

Simple path expression:            Result:

  /blog/post/title                   <title>Blogging</title>
                                     <title>W3C Standards</title>

Path expression with //:           Result:
  /blog//title                       <title>My new blog</title>
                                     <title>Blogging</title>
                                     <title>W3C Standards</title>

Path expression with predicate:    Result:
  /blog/post[@ID="p81"]/content      <content>Test content</content>

Path expression with filter:       Result:
  /blog/post/fn:id("p94")/title      <title>W3C Standards</title>
```

tree model as XPath does. The transformation logic of XSLT can be separate into reusable templates, which can be called like functions in procedural programming languages. These templates are associated with patterns, which match the nodes in the input tree (see Example 1.7). When the processor

starts the transformation, it firstly looks for the template matching the root node. Inside a template we can specify, which nodes we want to be processed next. After identifying a template for the specified node processor executes it. This terminates when processor reaches a template without other explicitly called templates. As a result we get a document composited from parts created by templates.

XSLT defines built-in templates, which can be overridden by user. For the root node and element nodes the built-in template calls `apply-templates` (see next paragraph) to continue processing all child nodes. For attribute and text nodes, the built-in template outputs the node value only. For other node types, nothing is done.

There is plenty of XSLT elements described in [W3C07]. There are structural elements, flow-control elements or, e.g., conditional elements. Here we introduce some of them briefly:

- `transform` – top-most element in XSLT document
- `template` – defines a new template rule with the specified pattern
- `param` – declares a parameter
- `variable` – defines a variable
- `apply-templates` – processes each node in the identified node set
- `call-template` – invokes a template by name
- `for-each` – iterates through the identified node set
- `sort` – sorts the current node list
- `choose` – selects one template among a number of possible alternatives
- `if` – tests a condition
- `copy-of` – copies the specified object
- `value-of` – generates a text node from an expression

1.4.4 XQuery

The *XML Query language* can be simply described by one sentence: it is to XML what *Structured Query Language* (SQL) is to database tables. Its main purpose is querying XML data. XQuery is an extension of XPath – it uses the same expression syntax and tree model [W3C10c].

Example 1.7 XSLT – transformation of XML from Example 1.6

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">
  <xsl:template match="/blog">
    <publishedPosts>
      <xsl:apply-templates select="post">
        <xsl:sort select="title" />
      </xsl:apply-templates>
    </publishedPosts>
  </xsl:template>
  <xsl:template match="post" >
    <post>
      <heading><xsl:value-of select="title" /></heading>
        <xsl:if test="@accessCount &gt; 200">
           <views><xsl:value-of select="@accessCount" /></views>
        </xsl:if>
    </post>
  </xsl:template>
  <xsl:template match="." />
</xsl:stylesheet>

Result:
  <publishedPosts>
    <post>
      <heading>Blogging</heading>
    </post>
    <post>
      <heading>W3C Standards</heading>
      <views>234</views>
    </post>
  </publishedPosts>
```

Queries in XQuery are very similar to queries in SQL – they are organized into *FLWOR* (pronounced "flower") expressions comprising these five clauses: `for, let, where, order by, return` (see Example 1.8). FLWORs, unlike path expressions, allow to manipulate, transform and sort results.

for This clause creates an iteration through the `post` nodes in Exam-

Example 1.8 Simple FLWOR expression

```
<posts>
  {for $post in /blog/post
   let $views := data($post/@accessCount)
   where $views > 200
   order by $post/title
   return <views>{$views}</views>}
</posts>
```

ple 1.8. The rest of the FLWOR expression is evaluated once for each of the posts. Each time a variable named `$views` is bound to a different atomic value of `accessCount` attribute. Dollar signs are used to indicate variable names in XQuery.

let The `let` clause is used to set the value of a variable. In Example 1.8 the `let` clause assigns an atomic value of `accessCount` attribute to a variable called `$views`. The `$views` variable is then referenced later in the FLWOR, in the `return` clause. The `let` clause serves as a programmatic convenience that avoids repeating the same expression multiple times. Using some implementations, it can also improve performance, because the expression is evaluated only once instead of each time it is needed [Wal07].

where This clause filters the nodes on a boolean expression. In Example 1.8 it selects only those `post` elements, which `accessCount` attribute value is higher than 200. This has the same effect as a predicate `[@accessCount>200]` in a path expression.

order by This clause sorts the results by post title. Sorting is an additional functionality to path expressions.

return This clause indicates return of the results. In Example 1.8 it returns XML snippet with post views. The curly braces around, e.g., the `$views` variable signify that it is an expression that is to be evaluated.

A path expressions in FLWOR may return a sequence of nodes, where some of them are repeated. In that case we can use a `distinct-values` function to get only unique nodes, not duplicates. Aggregate functions as `sum, count, min, max, avg`, which are well-known from SQL, are provided by XQuery too. What is not provided in the current version of XQuery 1.0, is a `group by` clause. It is planned in a next version 3.0 [W3C11a] (renamed from XQuery 1.1 to align with the family of "3.0" specifications). Until that time, aggregate queries must be written using nested FLWOR expressions.

Joins in XQuery are specified similarly to joins in SQL. See Example 1.9. There is an XML document containing users and teached courses. Below it there is a FLWOR expression, where we select users, who teach the course Modern Approaches. Since an `ID` attribute value from an `user` element is present also in a `role` element as an `uid` attribute value, it is used to join these two sources together through a predicate `[@ID=$role/@uid]`.

XQuery supports a lot of built-in functions, which origin is in XPath. For example there are functions for string conversions, a date and time comparison, a node manipulation, a sequence manipulation, boolean values and many more. It supports also user-defined functions, which is very useful for long queries called more times.

1.4.5 XQuery Update facility

XQuery language can query an input document, but it does not have a specific expressions for inserting, updating or deleting XML data in a database. This gap is filled by *XQuery Update facility*, which is an extension of XQuery. This language provides facilities to perform the following operations [W3C11b]:

- insertion of a node
- deletion of a node
- modification of a node by changing some of its properties while preserving its node identity
- creation of a modified copy of a node with a new node identity

Example 1.9 FLWOR expression using join

```
Queried XML document:
<?xml version="1.0"?>
<school>
  <courses>
    <course name="Modern Approaches">
      <role uid="u33">course-teacher</role>
      <role uid="u34">course-teacher</role>
    </course>
  </courses>
  <users>
    <user ID="u33">
      <nick>nick76</nick>
      <realName>Nicky Nick</realName>
      <lastName>Nick</lastName>
    </user>
    <user ID="u34">
      <nick>sue34</nick>
      <realName>Suzie Sue</realName>
      <lastName>Sue</lastName>
    </user>
  </users>
</school>

FLWOR expression:
<teachers course="Modern Approaches">
  {for $role in
     /school/courses/course[@name = "Modern Approaches"]/role
   let $user := /school/users/user[@ID=$role/@uid]
   order by $user/lastName
   return
      <teacher>
        <name>{data($user/realName)}</name>
        <nick>{data($user/nick)}</nick>
      </teacher>}
</teachers>
```

Chapter 2

XML databases

2.1 Background

In these days people use an XML format for exchange data more and more. The basic examples are web services *Simple Object Access Protocol* (SOAP) and *Representational State Transfer* (REST), which are widely used for communication and exchange data between various applications and systems over the Internet, where is no other way how to simply send and receive the data in a universal readable format.

Current applications usually use relational database systems, which require transforming of XML data to an appropriate format before storing them to the database. This step can be easily omitted if the application uses an XML database. This is also the main advantage of XML databases. Nowadays, when information systems are huge and work with different platforms and structures, it is important to communicate in a universal language, which every application understands and is able to handle it. XML databases help applications to remove the need of additional data transformation and save some XML processing time this way.

XML databases work with both document-centric and data-centric documents (mentioned in Section 1.2.2). Characterizing documents, that will be stored in a database, is crucial for correct choice of XML database type, which

will be used in an application. The *XML:DB initiative* has defined two different types of XML databases: *Native XML database* (NXD) and *XML Enabled database* (XEDB).

Definition 2.1.1 ([Ini03]) *Native XML Database*

1. *Defines a (logical) model for an XML document – as opposed to the data in that document – and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX 1.0.*
2. *Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.*
3. *Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.*

Definition 2.1.2 ([Ini03]) *XML Enabled Database is a database that has an added XML mapping layer provided either by the database vendor or a third party. This mapping layer manages the storage and retrieval of XML data. Data that is mapped into the database is mapped into application specific formats and the original XML meta-data and structure may be lost. Data retrieved as XML is NOT guaranteed to have originated in XML form. Data manipulation may occur via either XML specific technologies (i.e., XPath, XSLT, DOM or SAX) or other database technologies(i.e. SQL). The fundamental unit of storage in an XEDB is implementation dependent.*

The fundamental unit of storage in an XML database is equivalent to a row in a relational database. We can retrieve smaller units of data from the database (i.e., document fragments or individual structural components as elements, attributes, etc.) similarly to a relational database, where we can retrieve single columns from a row.

The main difference between NXD and XEDB is that XEDB does not store data in XML format. It uses only XML mapping layer, with which we can communicate with XEDB like with NXD. Under this mapping layer is other structure (relational, object-oriented, etc.), which stores the data, so there is always a need to convert data from XML format to appropriate underlying structure format. This conversion can cause a loss of some information from the original XML document (i.e., comments, processing instructions), so it is important to decide correctly, whether this is acceptable for an application using the XML database or not.

Known NXDs are Sedna [fSPR], eXist [Mei00], BaseX [Tea05], eXtraWay [Inf]. Between XEDBs are such databases as eXtremeDB [McO], PostgreSQL [Gro], Orient ODBMS [Tec].

During the blog.matfyz.sk portal development my colleagues decided to deploy native XML database named Sedna [fSPR] on the portal. The main reason was to research a new technology in the real world and reveal the pros and cons of such technology compared with a relational database. In next sections we discuss NXD in more detail and introduce Sedna NXD.

2.2 Native XML database

2.2.1 Storage architecture

The architectures of NXDs fall into two broad categories: *text-based* and *model-based* [Bou05].

Text-based NXD

Text-based NXDs store XML as text. It can be for example a file in a file system, a BLOB (Binary Large OBject – a collection of binary data stored as a single entity in a DBMS) in a relational database or a text format. Crucial components of text-based NXDs are indices. An XML document can be easily traversed thanks to them. Such databases, when retrieving whole documents or document fragments, perform a single index lookup, position the disk head

once, and, assuming that the necessary fragment is stored in contiguous bytes on the disk, retrieve the entire document or fragment in a single read [Bou05]. Reassembling document from pieces requires more index lookups and disk reads. An example of a text-based database is eXtraWay XML [Inf], which stores XML document as a file in a filesystem.

Model-based NXD

This type of databases creates an internal model of an XML document and stores it. There are different ways of storing the model. One way is to use a relational or object-oriented database, other databases use proprietary storage formats optimized for their chosen data model. Model-based NXDs built on other databases have performance of retrieving documents similar to those databases because they rely on these systems during this process. Performance of model-based database depends on output format – text-based databases are faster at returning documents as text, whereas model-based databases are faster at returning documents as DOM trees, assuming their model maps easily to DOM. An example of model-based database is Sedna [fSPR], BaseX [Tea05], eXist [Mei00] (all use proprietary storage model), Ozone [Ozo] (uses object-oriented storage model).

2.2.2 Main features

In this section we describe important features of native XML databases.

Document collections

Document collections in NXDs have the same meaning as tables in relational databases. Collections are used to store XML documents with a similar structure. We can create, e.g., a users collection, where each user has its own document with personal info and published posts and execute statements over the whole collection at once.

Query and update languages

Every database needs to support a language, that allows to query and update data in the database. We introduced such languages in Section 1.4. For a long time there was not a universal update language recommended by the W3C organization. Developers working on their XML databases had to try to create and implement their own solutions of an update language, so nowadays there is a lot of variations of it. The W3C Recommendation of an update extension to XQuery named XQuery Update Facility 1.0 (see section 1.4.5) was released in March 2011. It is known, that some NXDs already support this language (i.e. BaseX [Tea05], eXist [Mei00]). The final decision about which database to use is on a user. The user has to consider his/her needs about statements (queries/updates) executed over the XML documents and how a supported languages satisfy them.

Indices

It is important to note that NXDs support a creation of indices to boost statement execution time. Their implementation vary with different databases and depends on a storage implementation. We distinguish two types of indices [Bou05]:

value index This type of index references all text nodes and attribute values of a document. It is used to boost an execution time of content-based statements. A classical example for an application of a value index is a path expression with equality predicate.

full-text index This index type indexes tokenized text nodes. It increase a performance of statements, which search certain words in text nodes.

Transactions, concurrency and locking

Every correctly designed NXD supports transactions, locking and rollbacks, but the locking level may vary. There are two basic types of the locking level in

NXDs – *document-level locking* and *node-level locking*. Document-level locking, as its name tells, causes locking of the whole document during the update execution. This can lead to low multi-user concurrency, since everytime, when two users want to update the same document, one transaction has to wait for the end of another. If update statements create significant part of the database load, it is more appropriate to choose a database, which uses node-level locking. This locks only updated node, not a whole document, so other user, who wants to update different part of the document, do not have to wait.

Application programming interfaces (APIs)

Many NXDs offer their own API in a programming language such as C/C++, Java, PHP, etc. API provides an interface for connection to the database, executing queries, updates, results retrieval and other tasks. Results are usually returned in a form of an XML string or a DOM tree. Although NXDs have their own APIs, two vendor-neutral APIs exist – *XML:DB API* from XML:DB.org [Ini03] (uses XPath as a query language) and *XQuery API for Java* (XQJ) [Cor09], which uses XQuery as its query language.

Round-tripping

Another important feature of NXD is a round-tripping. In this case it means that data retrieved from a database are in exactly the same form than original data inserted to the database. This feature is crucial for document-centric data mainly, since all structural components like comments, whitespaces or processing instructions form a significant part of an XML document. With XEDB, these information would be lost because of mapping XML format to underlying structure. In general, model-based NXDs support round-tripping of XML documents at the level of their document model. Text-based NXDs round-trip XML documents exactly. It depends on an application, which type of NXD suits it better.

Normalization

It is a basis of the relational theory. The main goal of normalization is to prevent an unnecessary redundancy in a database. Saving disk space is

then its additional asset. Normalizing data in NXD is largely the same as normalizing them in a relational database – the database design cannot contain any repeated data. It is appropriate to note, that this is not usually an issue for document-centric documents, where is a very small amount of common data.

Referential integrity

Referential integrity is closely related to normalization and signifies validity of pointers to related data. It is very important for maintaining a consistent state of a database. In the relational theory it means checking, whether a primary key tuple referenced by a foreign key really exists. In NXD it means checking, whether pointers used in XML documents refer to valid documents or their fragments. There are several forms of pointers in XML. DTD uses `ID` and `IDREF` attributes, as was mentioned in Section 1.3. Another way how to implement pointers in XML is *XML Linking Language* (XLink) [W3C10a], which creates and describes links between resources.

Scalability

NXDs use indices to make data retrieval effective like, e.g., relational databases do. Locating documents or their fragments then depends on index, not document, size. NXDs are therefore comparable to relational databases in data retrieval using the same indexing technology and we could say they scale as well as other types of databases. Actually, that does not have to be a true assertion always. Since NXDs are a relatively new type of a databases compared with, e.g., relational databases, the development of implementation techniques is still in extensive progress. Therefore scalability can be significantly influenced by a selected NXD implementation. Other way, how to boost scalability, is correct usage of indices. Creation and using of indices has to be in balance with database load to reach a stable database performance. If the database is overindexed and considerably updated at the same time, statement execution time speeds up rapidly, since index tree reconstruction is needed after each update. The statement optimization is other appropriate step closer to balanced database performance.

Chapter 3

Sedna

Native XML database Sedna was created in 2005 by the Institute for System Programming (ISP) of the Russian Academy of Sciences [fSPR94] as a full-featured database system for storing large amounts of XML data. They implemented it from scratch to benefit from maximum freedom in developing proper desing principles for managing XML data. The main contribution of this system is a novel memory management technique and a schema-based clustering storage strategy efficient for both XML querying and updating.

3.1 Architecture

Sedna is implemented as a full-featured database system – it supports all traditional database services such as query and update facilities, external memory management, concurrency control, query optimization, etc. The decision to create the database from scratch was made because the ISP did not want to compromise because of any other existing database system. It took more effort, but they were able to design the system to their liking. Sedna uses XQuery and its data model (see Section 1.4) as a query language. For updating, ISP group implemented its own language extending XQuery named XUpdate (see Section 3.3). The main reason for XUpdate implementation was that in a database creation time there was no update language for XML data recommended by the W3C organization.

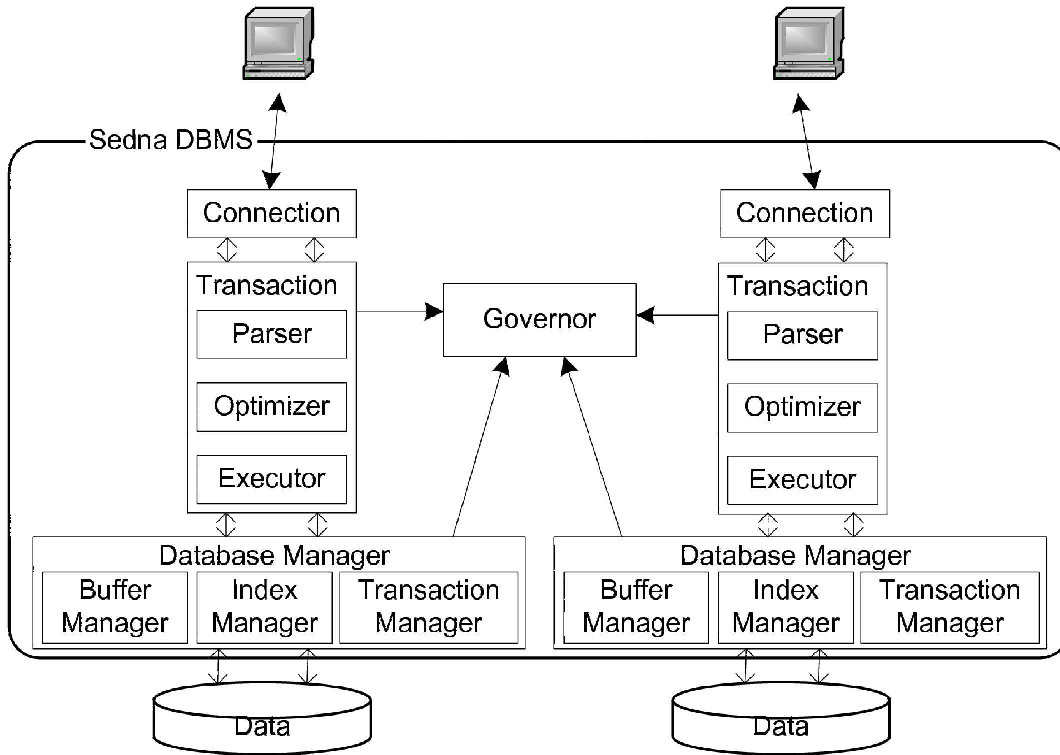


Figure 3.1: Sedna architecture [TSK+10]

The Sedna architecture (depicted on Figure 3.1) consists of these main components: *governor, connection, transaction* and *database manager* [TSK+10]. The governor is a control center of the whole system. It manages all databases and transactions running in the system. All other components are registered in it all their running cycle.

The governor creates an instance of the connection component for each Sedna client. The connection component encapsulates the *client session.* For each time, when the client initiates a transaction, the connection component creates an instance of the transaction component. This component encapsulates other three components involved in query execution: *parser, optimizer* and *executor*. The parser translates a query into its *logical representation* (a tree of operations), the optimizer then produces the optimized *query execution plan* from it, what is a tree of low-level operations over physical data structures. This execution plan is then interpreted by the executor.

On the physical level, each database is encapsulated by an instance of the *database manager*, which consists of the *buffer manager* that is responsible for the interaction between disk and main memory, the *transaction manager* that provides concurrency control facilities and the *index manager*, which is responsible for managing of indices [TSK+10].

3.2 Storage system

3.2.1 Data organization

Data organization was designed to provide an efficient execution of both queries and updates. These two main design strategies were used: *direct pointers* and *descriptive schema-driven storage strategy.*

Direct pointers

They are used to represent XML node relationships such as parent, child and sibling ones. Unlike relational-based approaches that require performing joins for traversing an XML document, traversing in Sedna is performed by simply following a direct pointer [TSK+10].

Descriptive schema-driven storage strategy

This is a novel storage strategy created by Sedna developers. It consists of clustering nodes of an XML document according to their positions in the descriptive schema of the document [TSK+10]. Descriptive schema is generated from data dynamically (and maintained incrementally) and creates a precise structure summary for data. It is in contrast to prescriptive schema (i.e., DTD), which is known in advance. Since descriptive schema is updated after changes in XML documents, it is more accurate than prescriptive one and it is possible to apply it to any XML document – even the one without prescriptive schema.

Descriptive schema serves as an entry point to the structural part of the XML document. Namely, every schema node has a pointer to data blocks that

store XML nodes corresponding to the given schema node. As XQuery queries and XML update statements access nodes in an XML document with XPath expressions, the descriptive schema plays a role of a naturally built index for evaluating XPath expressions [TSK+10]. Example of a descriptive schema is in Figure 3.2. Decriptive schema is presented as a tree of schema nodes, with each node labeled with an XML node kind (i.e., `element`, `attribute`, etc.). This node has a pointer to bidirectional list of data blocks storing XML nodes corresponding to this schema node. The structural part of a node reflects its relationship to other nodes (i.e. parent, children, sibling nodes) and is presented in the form of *node descriptor*. Each node descriptor is supplied with a *node handle*, that uniquely identifies the XML node in the database, provides quick access to it and stays immutable during the whole lifetime of the XML node.

3.2.2 Memory management

To benefit from usage of query execution logic to control the page replacement procedure (swapping) when forcing pages to disk, they decided to implement their own memory management mechanism that supports 64-bit address space and manages page replacement – *Sedna Address Space* (SAS) [TSK+10]. The key idea of memory management in Sedna is integrating persistence with virtual memory system. For achieving this integration, they divided database address space (DAS) into layers of equal size that fits virtual address space of a process (PVAS). A layer consists of pages; pages store XML data and also have equal sizes for uniform handling by the buffer manager. The 64-bit address of an object in DAS consists of the layer number (the first 32 bits) and the address within the layer (the remaining 32 bits). An address within the layer is mapped to the address in PVAS on the equality basis: the address of an object in the PVAS is the address of the object within the layer. The address range of PVAS is in turn mapped onto main memory by the Sedna buffer manager. Such address mapping allows dereferencing the pointer more effectively, as it eliminates pointer swizzling overhead. The main advantages of SAS used in Sedna are as follows [TSK+10]:

Example 3.1 A sample XML document

```
<library>
  <book>
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
  </book>
  <book>
    <title>An Introduction to Database Systems</title>
    <author>Date</author>
    <issue>
      <publisher>Addison-Wesley</publisher>
      <year>2004</year>
    </issue>
  </book>
  ...
  <paper>
    <title>A Relational Model for Large Shared Data Banks</title>
    <author>Codd</author>
  </paper>
</library>
```

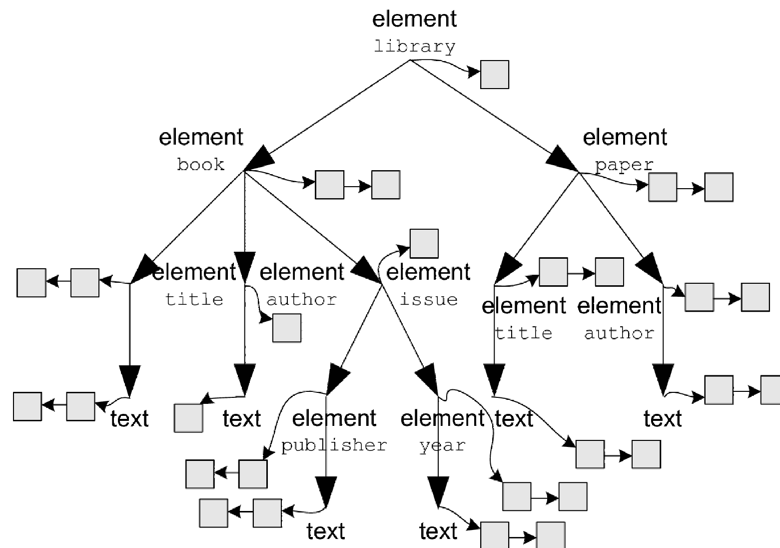


Figure 3.2: An internal representation of a sample XML document used in Example 3.1 in Sedna

- There is virtually no restriction on the database size, since it provides a 64-bit virtual address space, that can be emulated on a standard 32-bit architecture.
- Overhead for dereferencing a database pointer is comparable to the one for conventional pointers, since a database layer is mapped to PVAS addresses on equality basis.
- Costly pointer swizzling is avoided by using the same pointer representation in main and secondary memory.

3.2.3 Index management

Indices in Sedna are implemented as B+ trees, which are used often in relational databases also. In contrast to relational databases, the query executor in the current Sedna version [fSPR] does not use indices automatically. To enforce the executor to employ indices, specified functions have to be used [fSPR11b].

Example 3.2 Value index usage in Sedna

```
Statement to create an index:

  CREATE INDEX "postID"
  ON /blog/post BY @ID
  AS xs:string

Statement to use an index:

  index-scan("postID","p81","EQ")/content
```

Currently, Sedna supports two types of indices: value indices and full-text indices (see Section 2.2.2). In next paragraph we describe usage of value indices in detail. For detailed information about full-text index usage see [fSPR11b].

Value indices

In Example 3.2 we see, how the value index is used in Sedna. Firstly, it has to be created. The creation statement consists of these parts:

- index name (in our case `postID`)
- a path expression that identifies the nodes of a document or a collection that are going to be indexed (in our case `/blog/post`)
- a path expression that specifies the relative path to the nodes whose string-values are used as keys to identify the nodes returned by the first path expression (in our case `@ID`)
- an atomic type of a node used as a key (in our case `xs:string`)

Secondly, an index has to be explicitly used in a query or update statement. An index usage statement contains:

- a name of the index that we want to scan (in our case `postID`)
- a search value of the key of used index (in our case `p81`)
- a mode – it can be "EQ" (equal), "LT"(less than), "GT" (greater than), "GE" (greater or equal), "LE" (less or equal) (in our case `EQ`)
- nodes, which we want to retrieve (in our case `content`)

3.3 Update language

As we said earlier, Sedna team created their own update language named XUpdate. In this section we describe it in detail.

XUpdate extends XQuery language and is based on the XQuery update proposal by Patrick Lehti [Leh01] with the number of improvements. XUpdate provides these update statements: `INSERT, DELETE, DELETE_UNDEEP, REPLACE` and `RENAME`. The syntax and semantics of these statements are described in the following paragraphs (see Example 3.3 for the statement usage).

Insert statement

The `INSERT` statement inserts result of the given expression at the position identified by the `into` clause. In Example 3.3 the `INSERT` statement inserts the element `comment` to `comments` element.

Example 3.3 Usage pf update statements in Sedna

```
Insert statement:
  UPDATE insert <comment>First comment</comment>
    into /blog/post[@ID="p81"]/comments

Delete statement:
  UPDATE delete /blog/post[@ID="p33"]

Delete_undeep statement:
  UPDATE delete_undeep /blog/post//comment[@ID="p50"]

Replace statement:
  UPDATE replace $content in /blog/post[@ID="p81"]/content
  with <content>{data($content)} - content updated</content>

Rename statement:
  UPDATE rename /blog/post/title on heading
```

Delete statement

The `DELETE` statement removes persistent nodes from the database. It contains a subexpression, that returns the nodes to be deleted. In Example 3.3 this statement deletes a `post` element with certain `ID` attribute.

Delete_undeep statement

The `DELETE_UNDEEP` statement removes nodes identified by an expression, but in contrast to the `DELETE` statement, it leaves the descendants of the nodes in the database. In Example 3.3 it deletes undeep a `comment` element with certain `ID` attribute and assigns all its descendants to a parent of the deleted node.

Replace statement

The `REPLACE` statement is used to replace nodes in an XML document. In Example 3.3 it iterates over all the nodes returned by the `/blog/post[@ID="p81"]/content` expression, binding the variable `$content` to each node. For each binding the

result of the `<content>data($content) - content updated</content>` expression is evaluated.

Rename statement

The `RENAME` statement is used to change the qualified name of an element or attribute. In Example 3.3 it renames a `title` element to a `heading` element.

3.4 Transactions

A transaction can contain several statements, but by default it contains only one statement (in Sedna it is called an autocommit mode). It provides ACID (Atomicity, Consistency, Isolation, Durability) support during the transaction execution. In this section Sedna transaction issues are discussed briefly.

3.4.1 Multiversioning

This widely used technique for concurrency control management is also used in Sedna. The main principle of multiversioning in general is that each write operation on a data item `x` causes a generation of a new copy (version) of `x`. A database manager keeps this list of versions. The main contribution of this approach is that a scheduler does not have to reject transactions, which want to read already overwritten data. It just gives them slightly obsolete version of data. This results into less rejected operations and boosts the performance of the database. To control used storage space, old versions must be periodically purged and this action has to be synchronized with respect to active transactions, which use old versions of data [BHG87].

Sedna uses *Dynamic finite versioning* (DFV) [lWYsC93]. It uses snapshot-based scheme with data elements being pages. Snapshot is a set of versions (one version per page) that is transaction-consistent. Old versions are purged when they are not needed anymore, i.e., when they do not belong to any of the snapshots. Versioning mechanism is encapsulated in the storage manager [TSK+10].

3.4.2 Locking

Sedna uses the classical strict two-phase locking approach (S2PL) [BHG87] to support the isolation property of transactions. This approach allows multiple users access data without paying attention to concurrency mechanism details. Currently in Sedna, locking granularity is a whole XML document. In many cases, locking the whole XML document is excessive and leads to a decrease in concurrency [TSK+10]. This is the main reason, why Sedna developers work on a finer-granularity locking scheme.

3.4.3 Read-only transactions

Multiversioning allows using read-only transactions (also called queries). These transactions cannot contain update statements, but they can be executed much faster due to multiversioning. Each query reads one of the snapshots, so it obtains a consistent, but possibly a slightly obsolete state of the database [TSK+10].

In Sedna, read-only transactions are not used implicitly. During each client session there has to be an explicit function call, which turns on this read-only mode. After that, each statement of running transaction of that session is considered as a read-only statement.

3.4.4 Recovery

Durability property of transactions is guaranteed by logging and recovery mechanisms. All the main operations (insert node, create index, etc.) are logged. Additionally, a checkpoint may be created at some moment during execution to fixate transaction-consistent state of a database. This is called a *persistent snapshot*. If a database is crashed at some moment in time, recovery process is initiated to restore all transactions that had been committed by the moment of the crash.

Chapter 4

Optimization

First sections in this chapter talk about the Sedna optimization: why we need it, what are the hardware and software resources used on a web server, what Sedna settings are used. Next sections describe what has been done for the database optimization already and what we can do as a next step to boost statement execution time and to prevent database crashes. In the final section we discuss what can be used for a measurement of optimization results.

4.1 Why the database needs an optimization

In previous years students who worked on the development of the portal, had thesis assignments different from the database optimization. Everybody who managed the database or wrote the database statements, worked as the best as he knew, but nobody had the primary goal to take care of the optimization of the statements or the database settings. Bunch of students rotated over years during development this portal and the result was the requests on the database took too much time, we counted it in seconds. The database had been crashing from time to time also, especially during the end of the semester, when a lot of students needed to submit their course projects at once. This had been leading to a report creation and sending it to Sedna developers for research. In most cases they found out the problem, fixed it and released a new Sedna version. But constant iteration of these situations led to decision

to optimize our own Sedna instance and database statements too.

4.2 Hardware and software analysis

4.2.1 Server analysis

The web server, on which the portal is deployed, has a CPU with a model name AMD Athlon(tm) 64 X2 Dual Core Processor 3800+. It has two cores, each of them has a 2000 MHz clockrate and uses a 512KB L2-cache. An operation system on the server is GNU/Linux with Linux kernel version 2.6.34-hardened-r6. The RAM has a capacity of 2 GB and there are two harddisks, each has a capacity of 320 GB.

4.2.2 Sedna analysis

Settings analysis

Sedna version currently deployed at the portal is 3.5.161 for x64 platform. This is the newest version at this time [fSPR]. PHP API used with Sedna database has a version 2.3 and it is the newest version also.

Sedna has these main components (as described earlier in Chapter 3) – the governor, the database manager and the connection component. These components are implemented as a set of operation system processes and there are various options, which can be set at a process running. This is a complete list of Sedna processes in an alphabetical order and a brief description what they do:

- `se_cdb` - create a database
- `se_ddb` - delete a database
- `se_exp` - provide functionality of importing/exporting data
- `se_gov` - start governor
- `se_hb` - make a hot backup of a database
- `se_rc` - get the information about which components are run
- `se_sm` - run a database

- `se_smsd` - stop a database
- `se_stop` - stop governor
- `se_term` - interactive terminal to Sedna
- `se_trn` - session process

Every process except `se_trn` can be run from a command line with some options. For `se_gov` and `se_cdb` processes these options can be stored in an XML configuration file, so they do not have to be typed with every process running. Sedna Administration Guide [fSPR11a] describes, which options can be set with which process in detail and also says, what are the default values of these options.

PHP API analysis

PHP API is the Sedna Database driver implemented as an extension module for PHP. This driver offers a number of functions which are used for communicating with the database. Between the standard functions (connect to the database, close connection, execute statement, ...) [fSPR10] there is also one function `sedna_tweak_opt`, which can specify session options.

Here is a list of the `sedna_tweak_opt` function options and their description [fSPR10]:

- `SE_OPTID_HOST` - name of the host connected to
- `SE_OPTID_DATABASE` - name of the database connected to
- `SE_OPTID_USER` - user name
- `SE_OPTID_PASSWORD` - password
- `SE_OPTID_DIRECTORY` - what directory is searched for XML files, when executing LOADs
- `SE_OPTID_AUTOCOMMIT` - whether autocommit mode is enabled - default value is `true`
- `SE_OPTID_READONLY` - whether readonly mode is enabled - default value is `false`
- `SE_OPTID_QUERY_TIMEOUT` - query execution timeout in seconds - default value is `0` (means infinity)

- `SE_OPTID_MAX_RESULT_SIZE` - limit on the query result size - default value is `0` (means infinity)
- `SE_OPTID_LOGLESS` - whether logless mode is enabled - default value is `false`

In Section 4.4.2 we describe, which options are interesting to us and why.

Since students, who worked on the development of this portal, had work assignments different from the database optimization, nobody has ever changed the default option values of the processes or PHP API functions mentioned above.

4.2.3 Our database structure

Our database structure displayed on Figure 4.1 was designed by one of my colleagues Anton Kohutovic, who created blog.matfyz.sk portal from scratch in 2008 [Koh08]. An XML document conformed to this data model is created for each registered user and all registered users are collected in a one collection called *weblog*. A root element of this XML document is `user`, all other elements are its descendants. User's information stored in the XML is divided into three main parts. User's login information is stored in an element `account`, user's personal data are stored in an element `info` and all blog data, such as posts and comments are stored in an element `blog`. A depth of the XML tree of some user is not fixed, as the element `comment` may contain arbitrary number of other `comment` elements. Currently, the maximal tree depth is 52. Each `user`, `post` and `comment` has a unique attribute `ID`. User's `ID` attribute has a form of `'u'+NUMBER`, post and comment `ID` attribute has a form of `'p'+NUMBER`.

When a data model for a relational database is created, it is common to design, which items will be indexed. A designer assumes, these items will be frequently used to filter out data. In case of our data model, indices were not designed, so not implemented in the first phase of the portal development. As we will see later, this was a disadvantage of the database structure, which showed as statements with long execution time.

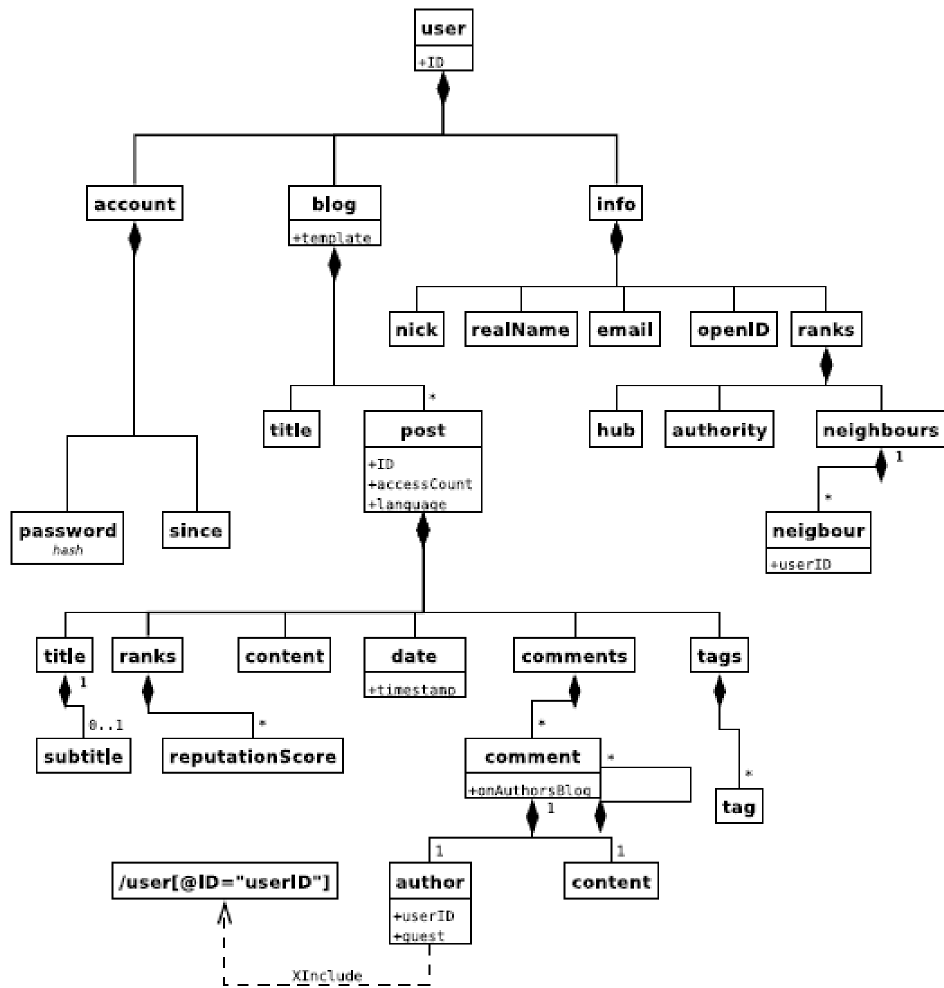


Figure 4.1: Data model scheme [Koh08]

4.3 Existing optimization

As we wrote in Section 4.1, the optimization of the database has never been the main goal of the students working on the portal. When problems with Sedna started, one student, Martin Rejda, who worked on modular redesign of the portal as his master thesis in 2010 [Rej10], tried to find out, what could be a cause of these problems. After his analysis of Sedna behaviour he discovered the problem was I/O wait, as Sedna uses a temporary file `database_name.setmp` for storing intermediate results retrieved during the statement execution. His solution was to store this file on a ramdisk. This change caused almost zero seek time as Sedna wrote its intermediate results directly to RAM, not to harddisk. But this adjustment did not solve all problems with Sedna totally and there was still a space for optimization [Rej10].

In Section 4.2.3 we talked about indices and their usage in our database structure. No indices were designed and implemented in the first phase of the portal development, but in 2011 two indices were added – `postID` and `tags` because of rank manager. It is used for an effective administration and indexing of tags used in blog posts [Ďu11]. As we made an analysis of these indices usage, we discovered `tags` index was not used at all in the database statements and `postID` index was used only in rank manager statements, so there was no performance improvement caused by the implementation of these indices in the rest of the portal.

4.4 How to optimize

4.4.1 Sedna settings

In this section we explain, which processes and their options could be attractive to us considering the optimization goal.

Since the optimization of the database requires starting from the base, first process, we are interested in, is `se_gov` (starts the governor). It has six options [fSPR11a], from which two we try to adjust during the optimization:

el-level level

It specifies the event log severity level. Value range is from 0 to 4 (4 means "Log everything", 0 means "Logging is off"). Default value is 3 (log all errors/warnings and all system messages - statements, resources statistics, ...). An optimization proposal is to set value to 2 (log all errors/warnings), since it decreases the amount of written information to harddisk.

alive-timeout timeout

It is a session keep-alive timeout. It specifies number of seconds to wait for the next request from some client on the same connection. Default value is infinite timeout. An optimization proposal is to set value to 1-3 seconds. Useless connections are then closed and reused for other clients. This option would be useless, if Sedna supported connection pooling. Then the infinity value would be desired.

Second process important to us is `se_cdb` (creates a database). This process has eleven options [fSPR11a], one of them we try to adjust:

bufs-num N

This option sets the number of buffers, with which the database manager can work in main memory. Default value is 1600 (64KB per buffer). An optimization proposal is to increase this value to 8000 buffers (500 MB), considering the RAM capacity in Section 4.2.2, since it enlarges the amount of database data located in main memory, which are then accessible quicker.

These optimization proposals (except `alive-timeout`, which will be used only on the live version of the portal, not during testing, since it would distort the optimization results) are used during the testing of Sedna performance and in the next chapters the results are presented.

4.4.2 PHP API function options

In Section 4.2.2 we described all the `sedna_tweak_opt` function options. This function adjusts client session options and since these options tweak the database performance, we consider them to be important.

Two of these function options we use also during the optimization, since they can affect the statement execution time in a significant manner.

SE_OPTID_AUTOCOMMIT
: This option is useful for merging more statements into one transaction, since by default every statement is executed in a single transaction. This option showed to be crucial for getting and setting the maximum `ID` of the elements `user, post, comment` (see Section 7.2.2).

SE_OPTID_READONLY
: This option guarantees that each statement, which is read-only (does not update any data), is able to read the data immediately, although maybe slightly obsolete and does not have to wait for a lock release (see Section 3.4.3). This option can be used every time, when a `select` statement is executed.

4.4.3 Statements optimization

This section talks about general tips for an XQuery statement performance boost [Wal07], which should be used during the statement optimization.

Avoidance of reevaluating the same or similar expressions

In a single statement, using a `let` clause for evaluating an expression and then binding its value to a variable, which will be referenced more times, is more effective than evaluating the expression itself many times. If there is a possiblity that some expression will be executed more than once, the `let` clause should be used instead. See Example 4.1.

Avoidance of unnecessary sorting

When creating a `FLWOR` expression, it is pertinent to know, if we are concerned about an ordering of the expression result or not. This decision can considerably boost the statement performance. See Example 4.1 (an inefficiency resides in sorting the two path expression, which are then re-sort in document order again because of union). Even if we want the result to be

sorted, there are some steps during evaluating that still can be declared as unordered to save some time. For detailed informations see [Wal07].

Avoidance of expensive path expressions

Path expressions, which use `descendant-or-self` axis (abbreviated `//`), can be very time consuming, because every descendant node has to be checked during the evaluation. If we know the whole path to the last descendant, is much more efficient to write it straight away than use `descendant-or-self` axis.

Usage of predicates instead of `where` clause

As written in the title, when using a `FLWOR` expression, is more effective to write the `where` clause as a predicate in the path expression. This filters out the elements before they are selected and evaluated in the next step of the `FLWOR` expression.

Usage of indices

Indices are very useful, if they are set appropriately considering the database structure and the database statement types. What does appropriately mean in this context – it is very important to analyse, which elements and attributes are used in the path expression predicates and analyse the statement types. After that we can decide what indices to set. There is a thin line between a well-optimized database using indices and overindexed database. If there is a lot of update statements on the database, indices can have the opposite effect on execution time as we assumed, since index tree needs to be updated too often.

4.5 Optimization results measurement

When optimizing the database, it is important to have a tool to measure a difference between its non-optimized and optimized version. Such a tool is called a benchmark.

Example 4.1 Usage of statement optimization techniques

```
Avoidance of reevaluating the same or similar expressions:
  Less efficient query:
   for $post in /blog/post
   where data($post/@accessCount) > 200
   order by $post/title
   return <views>{data($post/@accessCount)}</views>

  More efficient query:
   for $post in /blog/post
   let $views := data($post/@accessCount)
   where $views > 200
   order by $post/title
   return <views>{$views}</views>

Avoidance of unnecessary sorting:
  Less efficient query:
   let $blog := /blog
   return $blog//title | $blog//heading

  More efficient query:
   unordered{
    let $blog := /blog
    return $blog//(title|heading)
   }
```

Definition 4.5.1 *Benchmark is a program that is specially designed to provide performance measurements for a particular operating system or application.*

We use an XML database benchmark to measure, how the optimization boosts the database performance. The next chapter discusses existing XML database benchmarks and their characteristics – how they work, what they measure – and how these can be used for our purpose.

Chapter 5

Benchmarks

Definition 5.0.2 ([BMY09]) *An XML benchmark is a specification of a set of meaningful and relevant tasks, intended to assess the functionality and/or performance of an XML processing tool or system. The benchmark must specify the following:*

1. *a deterministic workload, consisting of a set of XML documents and/or a procedure for obtaining these and a set of operations to be performed*
2. *detailed rules for executing the workload and making the measurements*
3. *the metrics used to report the results of the benchmark*
4. *standard ways of interpreting the results*

5.1 Existing XML benchmarks

There are two main types of XML benchmarks. The first is an *application-level benchmark* and the second is a *micro benchmark* [Mly08]. The difference between them is that the former was created to compare different applications among each other, whereas the latter was created to study a given aspect of XML processing tool (e.g., performance, resource consumption, etc.).

In Table 5.1 we introduce six well-known benchmarks and describe their characteristics [Mly08][ABNE10][NKS07][RPJ+06].

	MBench	**XBench**	**XMach-1**	**XMark**	**TPoX**
Type of benchmark	Micro	Application-level	Application-level	Application-level	Application-level
Source	Synthetic	Synthetic	Synthetic	Synthetic	Synthetic
Number of users	single	single	single/multi	single	single/multi
Number of docs	1	Mixed	10^4 to 10^7	1	$3.6 * 10^6$ to $3.6 * 10^{11}$
Select statements	25	20	8	20	7
Update statements	5	0	3	0	10
Metrics	response time	response time	throughput	response time	throughput, resp. time

Table 5.1: Comparison of different benchmarks

MBench

This benchmark is a micro benchmark and uses single-document database scenario. The data used in the database are code-generated, but a user can set the size and the depth of the document. Contrary to other benchmarks in this comparison, this benchmark uses update statements also - inserting a node, deleting a set of nodes and bulk-loading of a new XML document.

XBench

This application-level benchmark also uses code-generated data. The database document can be selected from these four types: DC/SD, DC/MD, TC/SD, TC/MD, where DC is a *data-centric*, TC is a *text-centric*, SD is a *single document* and MD is a *multiple document* case. The size of the document can vary from small (10 MB), normal (100 MB), large (1 GB) to huge (10 GB).

XMach-1

Another application-level benchmark is XMach-1. This benchmark uses a web-based application scenario, so it is a first benchmark from this list, which can simulate multiple users. The number of the database documents vary from 10^4 to 10^7 with document size ranging between 2 KB to 100 KB. The maximum depth of an XML tree is restricted to 6. Since generated doc-

uments are very small, this benchmark is inappropriate for evaluating large scale implementations.

XMark

This application-level benchmark uses a single, code-generated document also, but contrary to other such benchmarks, with this benchmark user can set the document size with scaling factor. Although the document depth is fixed to 12, there are still more possibilities for document generation than in other benchmarks. The main drawback of this benchmark is its fixed depth and even distribution of elements at each level.

TPoX

Another application-level benchmark, which targets multi-user environments, uses code-generated data too. But in comparison to other benchmarks, the range of the document amount used in the database is the biggest - from $3.6 * 10^6$ to $3.6 * 10^{11}$. The document size varies from 3 KB to 20 KB. The tree depth and breadth is controlled by a template, which generates the data. This benchmark is the only one from this list, where the number of the update statements exceeds the number of the select statements.

5.2 Choosing an appropriate benchmark

In the previous section we mentioned known benchmarks and described their characteristics. In this section we discuss what benchmark features we required to measure optimization results and what benchmark, if any, was the most appropriate for our purpose.

From the base point of view, we needed to know, which type of benchmark we should use for a measurement of the optimization. It is clear that we needed a micro benchmark, since we studied only Sedna performance, not different applications among each other. As we can see in the table 5.1, there is only one such benchmark - MBench.

As we mentioned earlier (Table 5.1), every benchmark from our list uses

code generator for creating the database content. It means that there is no possibility to use own data as a database content. In our case, this feature was the most important for us, since we studied, how the database with original data reacted, when we changed the system settings or optimized the statements. From this point of view, no listed benchmark was pertinent for us.

The second very important feature for us was a usage of update statements in a benchmark. Since one of the essential actions on our portal is publishing posts and comments, we wanted to be capable of measuring the optimization results of update statements too. Other reason, why it was important for us, was that average execution time of update statements on the portal was very high before optimization and we needed to test the optimization contribution using a benchmark. In the list there are three benchmarks, which fulfilled this request. The TPoX benchmark emphasizes the update statements the most.

Another crucial benchmark feature for us was a multi-user testing. Since we had an enormous problem with the database each time, when a lot of users had practicals or submitted their projects, we needed to test, how the optimization impacted on this problem, by multi-user testing. There are only two benchmarks with this property - XMach-1 and TPoX.

We defined above, what benchmark features were significant for us and why. The next step was to choose the right benchmark. But here we encountered the problem, since there was no such an existing benchmark that satisfied all our requirements. The resolution was to create our own micro benchmark, which covered all mentioned features.

Chapter 6

Statement analysis

Since we did not have any statistics about which portal statements are executed the most or last the most time, we made a statement analysis. This chapter presents, how we gathered and analysed the data, and the analysis results.

6.1 Gathering the data

At the beginning, the most important information for us was, what statements are executed on the portal and how long they last. To find it out, we decided to store these five statement attributes after the statement execution – a complete statement text, its execution time in milliseconds, an execution endtime, a statement type (`select, update, insert, delete`) and a filename and a line, from where the statement execution was called. As a storage we used MySQL database, not our XML database. The main reason was, since we had problems with the XML database, we did not want to load it more than it was and cause another problems.

This statement logging was turned on twice and lasted one workweek each time. Since the portal is used mainly for the study purpose, the user activity is the highest during the semester and culminates at the end of the semester, since there are course deadlines for project submissions. This fact led us to decision to log an activity during the semester and at the end of the semester to see the difference between executed statement preferences.

	During the semester	At the end of the semester
Number of executed statements	514 322	1 878 849
Average statement execution time in ms	118.45	76.02
Average modification statement execution time in ms	348.47	637.77
Average `select` statement execution time in ms	108.69	65.96

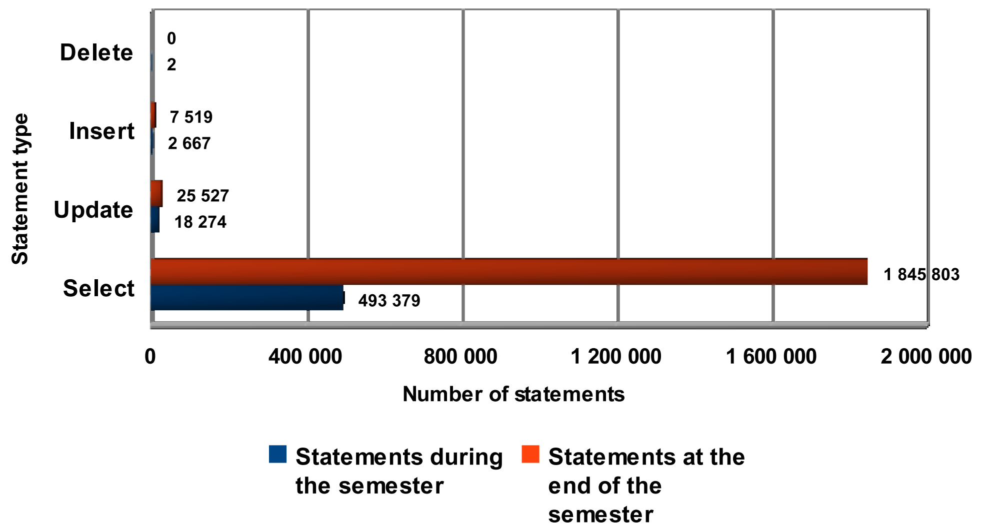
Table 6.1: Overall statistics

6.2 The data analysis

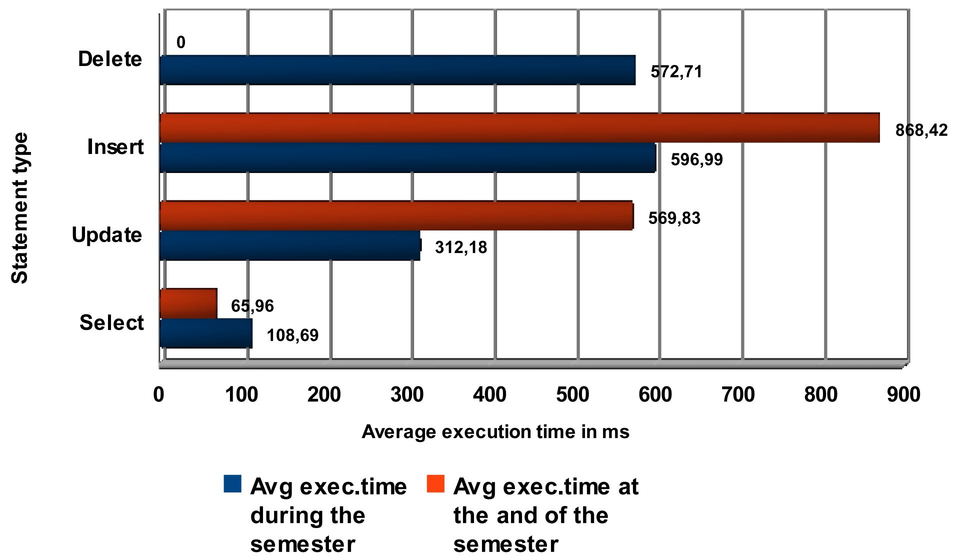
6.2.1 Overall analysis

After the data collection was finished, we started to analyse different aspects of the gathered data. Every aspect was examined particularly in during-semester statements and end-semester statements and the results were compared together. Overall statistics are in Table 6.1.

The first examined aspect was a number of executed statements of certain type. The results are shown on Figure 6.1a. As we assumed in the beginning, there was a significant difference between the load during and at the end of the semester. Modification statements (`update, insert, delete`) were executed about 55% more at the end of the semester than during the semester. When we compared `select` statements, an increase was even larger – more than 370%. A modification and a `select` statement ratio was 1:24 during the semester and 1:49 at the end of the semester. From these results we see clearly that the XML database was loaded mainly by reading the data, not by modificating them. Considering these results during the optimization, we decided to accept a little average execution time increase of modification statements caused by a rebuilting of an index tree, if a particular index boosted an average execution time of `select` statements significantly.



(a) Number of executed statements of certain type



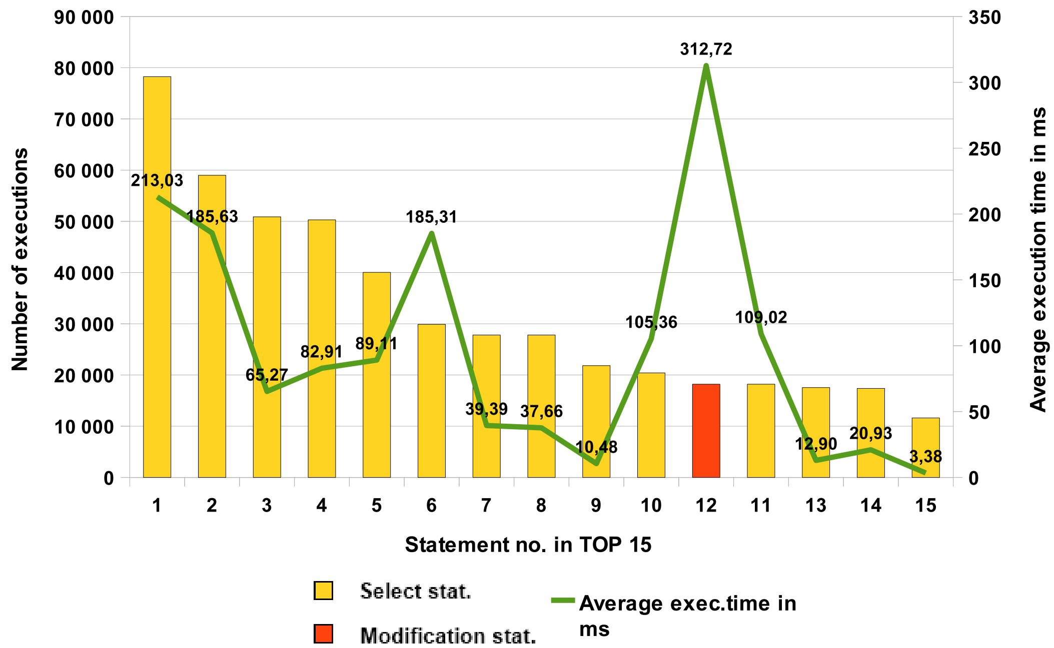
(b) Average statements execution time

Figure 6.1: Analysis of statement characteristics

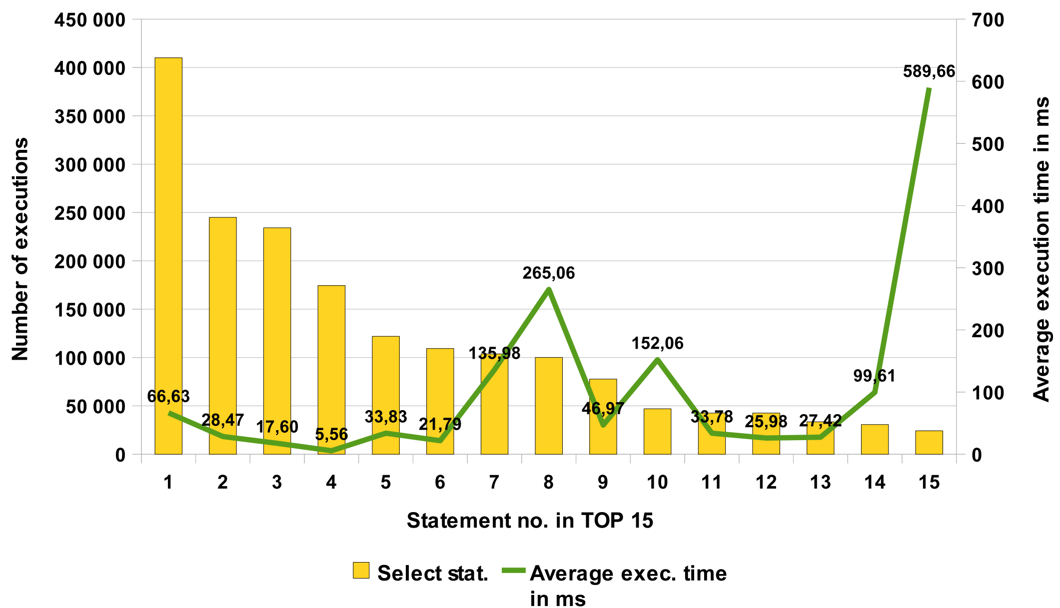
The second examined aspect was an average execution time of a certain statement type. The results are displayed on Figure 6.1b. We see that an average execution time of modification statements was much more higher than an average execution time of `select` statements. Concrete numbers say that the average execution time of modification statements in total was 348.47 milliseconds during the semester and 637.77 milliseconds at the end of the semester (see Table 6.1). Alarming was that in both logged periods an average execution time of every modification statement type was more than 300 milliseconds and an average execution time of every `select` statement type was more than 60 milliseconds, what was unacceptable for us.

6.2.2 Different statements usage

The next step was to survey, how many different statements were executed during both logged periods and how long they were executed. By different statements we mean statements, which calling location (a filename and a line) is different. The same statements are those, which were called from the same location. The analysis results of both logged periods are on Figure 6.2. Two subfigures depict top 15 the most executed statements with their average execution time during and at the end of the semester (these statements do not have to be the same, they can differ). As we see, the most executed statements had long average execution time also. There was only one modification statement on these figures – on Figure 6.2a. Another observation (see number of `select` statements on Figure 6.1a) revealed that these top 15 statements created more than 95% of the overall database load during the semester and 96% at the end of the semester. This means that the database performance should be boosted significantly by only optimizing these 30 statements (15 for both logged periods).



(a) Top 15 the most executed statements during the semester



(b) Top 15 the most executed statements at the end of the semester

Figure 6.2: Top 15 the most executed statements statistics

6.2.3 Predicates usage

When we knew the base statistical data (what statements were executed, how much, how long), we started to examine, which predicates were used the most in the statements and how long these statements were executed. This information was very important for us, since it helped us to choose the right elements and attributes from our XML structure as indices. On Figure 6.3b and 6.4b are displayed our results from the during-semester period and the end-semester period. These results contain all predicates used in the statements and say how much they were executed. We extracted these data from logged statement texts by matching a predicate pattern with a statement text.

From mentioned results we could already say, which elements and attributes are suitable candidates as indices, depending on their execution number. The index set would contain the elements and attributes of which predicates had execution number greater than 1 000 during the semester and greater than 100 000 at the end of the semester:

1. `user[@ID]` (`post[../../@ID]` is the same predicate)
2. `user[info/nick]`
3. `post[@ID]`
4. `post[@status]`
5. `post[@lang]`
6. `user[@ref]`
7. `course[@year]`
8. `post[ranks/reputationScore]`
9. `post[@accessCount]`
10. `post[@private]`

But there were cases, when some predicates were used only together with another ones, so we made also an analysis, which combinations were used the most. The results are depicted on Table 6.2.

As can be seen, there were certain predicates, which were used only in a combination with other predicates, e.g., `post[@status]`, `post[@lang]`, `course[@year]`

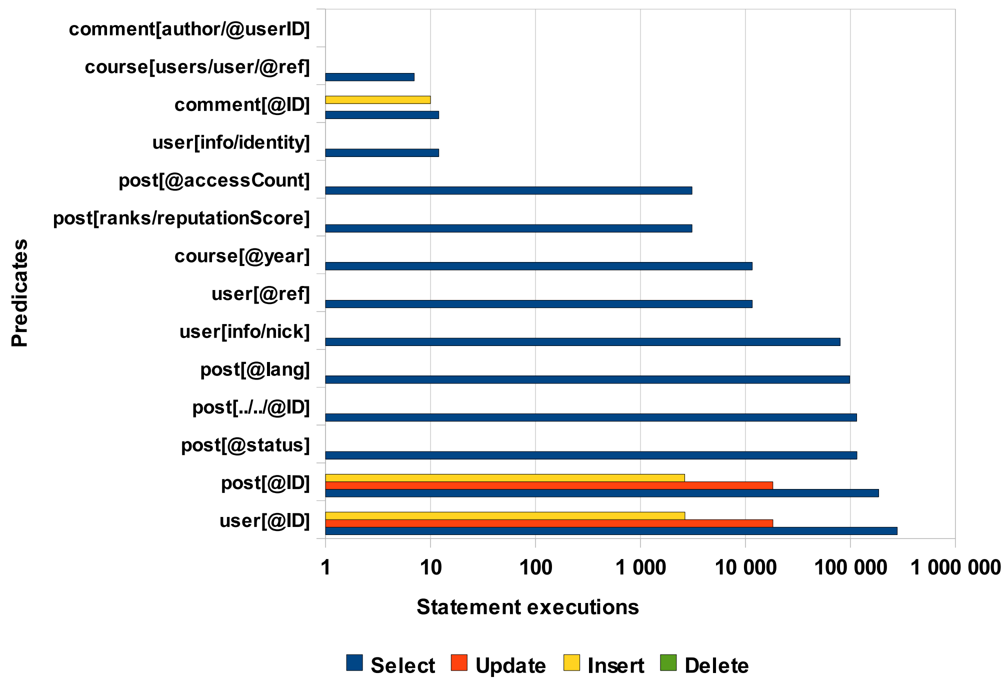
or `post[@private]`. Based on these analysis results, we decided to remove those attributes from the index set and adjust it to this form:

1. `user[@ID]`
2. `user[info/nick]`
3. `post[@ID]`
4. `user[@ref]`
5. `post[ranks/reputationScore]`
6. `post[@accessCount]`

This index set is used during the statement optimization described in the next chapter.

Used predicates	Statement			
	Select	Update	Insert	Delete
user[@ID]	279 529	18 274	2 655	1
post[@ID]	185 840	18 255	2 641	1
post[@status]	115 490	0	0	0
post[../../@ID]	114 897	0	0	0
post[@lang]	98 184	0	0	0
user[info/nick]	79 739	0	0	0
user[@ref]	11 631	0	0	0
course[@year]	11 631	0	0	0
post[ranks/reputationScore]	3 102	0	0	0
post[@accessCount]	3 102	0	0	0
user[info/identity]	12	0	0	0
comment[@ID]	12	0	10	0
course[users/user/@ref]	7	0	0	0
comment[author/@userID]	1	0	0	0

(a) Table depicted number of statement executions with certain predicate used

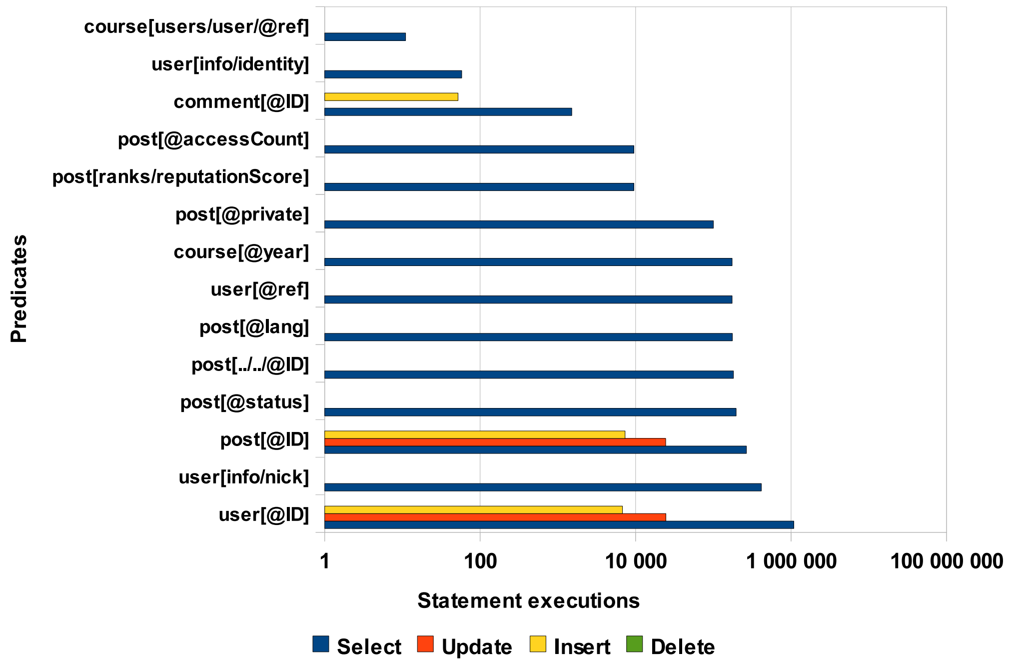


(b) Number of statement executions with certain predicate

Figure 6.3: Statistics retrieved during the semester

Used predicates	Statement			
	Select	Update	Insert	Delete
user[@ID]	1 083 970	24 515	6 779	0
user[info/nick]	414 480	0	0	0
post[@ID]	266 704	24 404	7 318	0
post[@status]	196 844	0	0	0
post[../../@ID]	181 462	0	0	0
post[@lang]	175 636	0	0	0
user[@ref]	174 403	0	0	0
course[@year]	174 403	0	0	0
post[@private]	100 130	0	0	0
post[ranks/reputationScore]	9 493	0	0	0
post[@accessCount]	9 493	0	0	0
comment[@ID]	1 512	0	52	0
user[info/identity]	58	0	0	0
course[users/user/@ref]	11	0	0	0

(a) Table depicted number of statement executions with certain predicate used



(b) Number of statement executions with certain predicate

Figure 6.4: Statistics retrieved at the end of the semester

Predicate combinations	Executions
user[@ID]	143 808
post[@status],post[@ID],post[../../@ID],post[@lang]	88 878
user[info/nick]	79 739
user[@ID],user[@ID]	57 415
post[@ID]	55 630
user[@ID],post[@ID]	41 821
post[@status],post[../../@ID],post[@ID]	20 408
course[@year],user[@ref]	11 631
post[../../@ID]	5 611
post[@lang]	3 102
post[@status],post[ranks/reputationScore],post[@lang]	3 102
post[@status],post[@accessCount],post[@lang]	3 102
comment[@ID]	22
user[info/identity]	12
course[users/user/@ref]	7
comment[author/@userID]	1

(a) Number of statement execution with certain predicate combination during the semester

Predicate combinations	Executions
user[@ID]	705 747
user[info/nick]	414 480
course[@year],user[@ref]	174 403
user[@ID],user[@ID]	121 938
user[@ID],post[../../@ID]	103 728
post[@status],post[@ID],post[@private],post[@lang]	100 130
post[@ID]	86 690
user[@ID],post[@ID]	61 861
post[@status],post[../../@ID],post[@lang]	47 027
post[@status],post[../../@ID],post[@ID]	30 701
post[@lang]	9 493
post[@status],post[ranks/reputationScore],post[@ID],post[@lang]	9 493
post[@status],post[@ID],post[@accessCount],post[@lang]	9 493
comment[@ID]	1 512
user[info/identity]	58
user[@ID],post[@ID],comment[@ID]	52
course[users/user/@ref]	11
post[../../@ID],post[@ID]	6

(b) Number of statement execution with certain predicate combination at the end of the semester

Table 6.2: Number of statement execution with certain predicate combination

Chapter 7

Portal benchmarks

In this chapter we describe in detail our benchmarks used for a measurement of the optimization results and the optimization results themselves. We used two different benchmarks, since we measured an optimization effect of the statement optimization firstly and then we measured the statement optimization effect altogether with the system and session options optimization effect by multi-user testing.

7.1 Our benchmark design overview

The most important part of our benchmark was a workload. We needed to examine different optimization techniques on our portal data, so our workload consisted of the real portal data binding to a specific date. Another important part of our benchmark was a set of statements to be performed. At the beginning we did not know, which statements to use, since we did not have any statistics, which statements are the most executed or last the most time. So we made a statement analysis during and at the end of the semester in the first place (see Chapter 6). Those statements, which were the most executed or lasted the most time, were used in the set of statements for benchmarks.

We created two different benchmarks. Each of them used the statement response time as a metric. First was *Simple benchmark*, whose purpose was to measure a statement and index usage optimization effect. The best optimized

statement versions were then used in our *Complex benchmark*. Its purpose was to measure, how the overall optimization (statements, system and session settings) boosted the response time of complete actions executed on the portal, e.g., homepage loading, user homepage loading, etc. We simulated multi-user environment – the certain amount of users was run in parallel and each user executed the specific set of actions. Then we compared the action response time before and after the optimization.

Both benchmark constructions and the optimization results are described in detail in next sections.

7.2 Simple benchmark

Simple benchmark was designed to measure, how the statement optimization boosts the average statement execution time. The first step was to run the original statements in an iteration and measure their average execution time. The second step was to run the optimized versions of these statements in an iteration, measure their average execution time and compare the results.

7.2.1 Specification

Here we specify a Simple benchmark design according to Definition 5.0.2.

Deterministic workload

The benchmark workload consists of the real portal data binding to a specific date. More precisely, the workload is comprised from the database backup binded to a specific date. There are 1077 XML documents, their average size is 12.91 KB. The database size in total is 14.58 MB and a maximal depth of an XML tree is 52. The database already contains two indices – `postID` and `tags`, as mentioned in Section 4.3. During the benchmarking, this set of indices changes, since we add there indices, which were proposed in Section 6.2.3.

Set name	Statements in total	Select statements	Modification statements
Original statements set	27	22	5
Optimized statements set	12	11	1
Index Statements set	24	19	5
IndexII Statements set	4	2	2

Table 7.1: Operation sets – characteristics

Set of operations

Operations, which are used in the benchmark, are statements selected from the statement analysis results according to their number of executions and their average execution time. To the benchmark statement set we picked those statements from the during-semester period, of which execution number exceeded 500 or those, of which an average execution time exceeded 100 ms and their execution number was greater than 10. From the end-semester period we chose the statements with execution number exceeding 1000 or those, of which an average execution time exceeded 100 ms and their number of executions broke 1000 times limit. During the benchmarking, we worked with four statements set (see Table 7.1).

The first set consists of every statement which satisfied conditions mentioned above. It has 27 statements, 22 of which are `select` statements and 5 of which are modification statements. We named this set *Original statements set*, since it contains statements in their original version used on the portal. A full version of Original statements set is in appendix in Listing A.1.1.

The second set contains optimized versions of those statements from Original statements set, which were able to be optimized according to the tips discussed in Section 4.4.3, except indices usage. There are 12 statements in total, of which 1 is a modification statement, others are `select` statements. This set we named *Optimized statements set* and its full version is in appendix in Listing A.1.2.

The third set consists of index optimized versions of such statements, which were able to be optimized by indices usage. We picked statement candidates either from Original statements set or, if we have already had their optimized

version, from Optimized statements set. This statements set named *Index Statements set* contains 24 statements, 5 of them are modification statements, the rest are `select` statements. A full set version is in appendix in Listing A.1.3. Index set used for the optimization of these statements includes these 6 elements/attributes (as proposed in Section 6.2.3):

- `user[@ID]`
- `user[info/nick]`
- `post[@ID]`
- `user[@ref]`
- `post[ranks/reputationScore]`
- `post[@accessCount]`

The fourth set consists of index optimized versions of such statements, which were able to be optimized by indices usage again, but the index set is different from the previous one. Candidates for index optimized version were picked either from Original statements set or, if we have already had their optimized version, from Optimized statements set. After creating the optimized versions of statements, we excluded those statements, which have already been in Index Statements set. Created set named *Index II Statements set* has 4 statements, half of them are modification statements. A full version is in appendix in Listing A.1.4. The set of indices used here includes these 5 elements/attributes:

- `user[@ID]`
- `user[info/nick]`
- `post[@ID]`
- `post[ranks/reputationScore]`
- `post[@status]`

The reason why we used two different index sets was, we wanted to compare, which index set boosted the database performance more. First index set followed from the statement analysis results in Section 6.2.3. In the second set, we removed the `user[@ref]` index, since it was used only in one statement in Index Statements set and replaced the `post[@accessCount]` index with

`post[@status]` index. The reason for replacement was that the former index was used (apart form other statements) in two statements, which retrieve and update `accessCount` attribute of `post` elements. This activity is executed each time, when a blog post is displayed to user. Since the `accessCount` index has to be rebuilt after each `accessCount` attribute update, we replaced this index with `post[@status]` index. This index was chosen, because the predicate `post[@status]` (similarly to `post[@accessCount]`) occured often in a combination with others (as mentioned in Section 6.2.3).

The statements included in the sets mentioned above contains keywords in their text, specifically in predicates and in attribute values. These keywords are:

- `###USER_ID###`
- `###USER_NICK###`
- `###POST_ID###`
- `###COMMENT_ID###`
- `###MAX_ID###`

These keywords serve as a hint, where to insert parameter values. This strategy helps us to guarantee, that there are different statemets executed during the benchmarking.

Rules for executing the workload

Here we specify, how the benchmark is used and how we make the measurements.

When running the benchmark, two options have to be set. The first is, which statements set is going to be used and the second is the number of iterations of a statement execution. Then the benchmark starts to iterate through the selected statements set. It picks one statement during each iteration and iterates its execution defined number of times. Before each statement execution, the keywords in the statement text are replaced by values from the XML file, where 50 portal users are listed with their `ID, NICK, POST_ID` and `COMMENT_ID`. The values are chosen according to an iteration number of the

statement execution. The keyword `###MAX_ID###` is replaced with an up-to-date maximum `ID`.

After the statement execution the benchmark saves a statement execution time in milliseconds, add this value to a total execution time and counts a minimal and a maximal execution time. After the end of the iteration of the statement execution, the benchmark computes an average execution time of this statement and a standard deviation. All these results are then written to a file.

Metrics

The only one metric used in Simple benchmark is an average statement execution time measured in milliseconds.

How to interpret the results

Since the benchmark metric is an average statement execution time, the interpretation of the results is based on a comparison the average execution time of the original statement version with the average execution time of its optimized versions. We want to achieve the best average statement execution time during the optimization, so that statement version, which has the best average execution time, is declared the best statement version.

7.2.2 The results

As we wrote in the benchmark specification, there were four different statements sets, which could be used during the benchmarking. We used these four sets with the iteration number set to 500. Each statement was then executed 500 times. The results of Simple benchmark are presented on Figure 7.1. All partial results are in appendix (see Figures A.1 and A.2).

As we see on Figure 7.1, for every original statement version except one (no. 15) we created at least one optimized version, which had the average execution time better than the original version. The statement no. 15 was a special case, since there were no optimization possibilities according to general optimization tips mentioned in Section 4.4.3. Changes we did, did not cause a statement

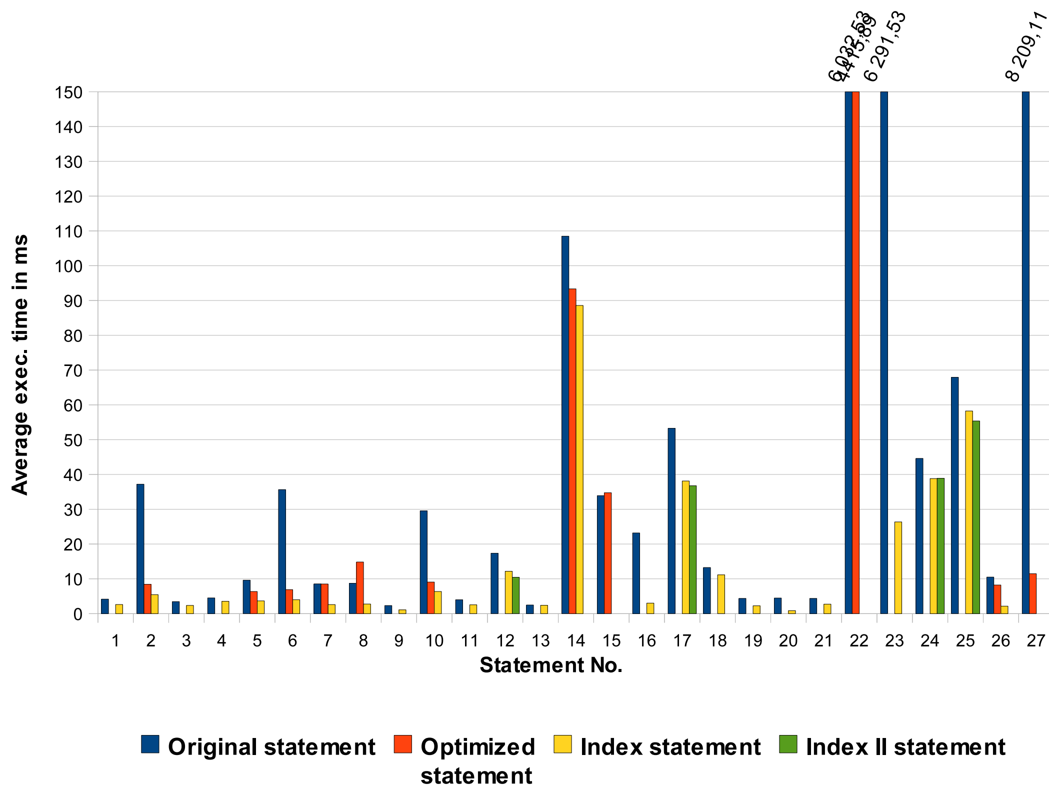
performance improvement. The average execution time of the statement no. 22 was improved from 6 032.53 ms to 4 415.89 ms. This execution time was still too high, but when we examined this statement in detail, we found out, this statement could be replaced by the statement no. 26. This replacement caused improvement from 6 032.53ms to 2.14ms. The long average execution time of the statement no. 23 was caused by a usage of `descendant-or-self` axis (abbreviated `//`). We were not able to get rid of it in the optimized version of this statement, since the `comment` element has no depth restriction for its subtree, but index usage boosted the statement performance enormously.

The statement no. 27 had the longest average execution time at all. This statement was used for finding the maximal `ID` attribute value used in `post` or `comment` elements, since we needed it during inserting the new `post/comment` element to the database. In relational databases, this functionality is provided by an auto-increment field – this allows a unique number to be generated when a new record is inserted into a table. In Sedna database, this functionality is not supported, so during the portal development, this had to be implemented by our colleagues. Created statement was searching the whole collection of users – every `post` or `comment` element in the `blog` element for finding the maximal used `ID`. Since its average execution time was so high, this caused also problems with uniqueness of a `post/comment ID` attribute. Sometimes happened there were 5 posts/comments with the same `ID`. Our solution was to change the way for getting a maximal used `ID` completely. We created an XML file, where the maximal values of `user` and `post/comment ID` attributes are stored. Now the operation of getting the maximal `ID` consists of two statements given into one transaction (first statement selects the maximal `ID`, second writes its new value to the database). This guarantees `ID` uniqueness in the whole database.

With this benchmark we examined also, which index set was more appropriate for our optimization purpose. The benchmark results showed, the second index set used in Index II Statements set was more appropriate for the optimization goal.

Statement ID	Statement Name	Original Statement	Optimized Statement	Index Statement	Index II Statement
1	Get user's ID	4.17		2.61	
2	Get 20 user's posts with certain tag ordered by timestamp	37.20	8.43	5.45	
3	Get user's blog title	3.46		2.36	
4	Get user's info	4.52		3.54	
5	Get blog customization	9.62	6.37	3.66	
6	Get the newest comment from user's posts	35.66	6.91	4.00	
7	Get certain post's title text	8.56	8.52	2.61	
8	Get post's user's nick	8.73	14.84	2.76	
9	Get user type	2.33		1.09	
10	Get all user's posts except one	29.54	9.08	6.36	
11	Get post accessCount	3.99		2.54	
12	Update accessCount in certain post	17.36		12.17	10.44
13	Get user type in course	2.49		2.40	
14	Get all user's posts	108.48	93.34	88.58	
15	Get users with at least one slovak post	33.91	34.75		
16	Get 10 best ranked published slovak posts	23.20		3.02	
17	Get 10 most read published slovak posts with access higher than 50	53.27		38.12	36.76
18	Create upload form for user	13.26		11.14	
19	Get post votes count	4.37		2.25	
20	Insert votes tag to post	4.48		0.87	
21	Get password text	4.38		2.72	
22	Check if comment with certain ID exists	6 032.53	4 415.89		
23	Insert comment to comment	6 291.53		26.36	
24	Insert post to blog	44.61		38.82	38.89
25	Update post	67.92		58.24	55.36
26	Check if post with certain ID exists	10.50	8.22	2.14	
27	Get max used ID	8 209.11	11.47		

(a) Table depicted an average execution time of statements used in Simple Benchmark



(b) Comparison of average execution times of all statements used in Simple benchmark

Figure 7.1: Simple benchmark results

7.3 Complex benchmark

Complex benchmark was designed to measure how the complete optimization (statement optimization, session and settings optimization) boosted the average execution time of whole actions executed on the portal, e.g., homepage loading, user's blog loading, post page loading, etc. It simulated multi-user environment – the certain amount of logged/not logged in users was run in parallel and each user executed the specific set of actions. Then we compared the action execution time before and after the complete optimization.

7.3.1 Specification

This section contains a Complex benchmark specification based on Definition 5.0.2.

Deterministic workload

There are two versions of the workload used in this benchmark. The first workload consists of the real portal data binding to a specific date. It is the database backup with the same characteristics as the workload of Simple benchmark (see Section 7.2.1). The second workload is the first workload with updated index set. It is that index set, which showed to be the best solution for the statement optimization in the Simple benchmark results (Section 7.2.2). The first workload is used with original statements, the second is used with optimized statements.

Set of operations

This benchmark contains two sets of operations. Each set consists of groups of statements, which represent basic actions executed on the portal. By the action we mean, .e.g., post page loading, homepage loading or post insertion. Both sets contain the same actions. Actions in the first set, named *Original actions set*, are created by original statements only (most of them were used in Simple benchmark operation sets too – Section 7.2.1). A full version of this set can be found on an enclosed CD. Actions in the second set, named *Optimized*

actions set, are created by the optimized versions of the statements used in Original actions set. Optimized statement versions are selected according to Simple benchmark results (Section 7.2.2). A full version of this set can be found on an enclosed CD. As mentioned in Section 7.2.1, statements can contain keywords, which are used for detecting a place, where to insert a parameter value. This is a complete list of keywords used in these statements and their description:

- `###USER_NICK###` – nick of a user, whose blog is viewed by other user
- `###USER_ID###` – ID of the user with `USER_NICK`
- `###USER_POST_ID###` – post ID belonging to the user with `USER_ID`
- `###USER_POST_COMMENT_ID###` – comment ID belonging to a post with `USER_POST_ID`
- `###LOGGED_USER_ID###` – ID of a logged in user
- `###LOGGED_USER_NICK###` – nick of the user with `LOGGED_USER_ID`
- `###LOGGED_USER_POST_ID###` – post ID belonging to the user with `LOGGED_USER_POST_ID`

Below is a complete list of actions used in the benchmark with their characteristics.

1. **Homepage**
   This action represents portal homepage loading.
2. **Homepage with tag**
   This action represents loading of the homepage with a specified tag. Tag is a keyword, with which the user can describe the content of his/her post. Homepage with a specified tag contains only those posts, which have this tag assigned to their content.
3. **User's blog**
   This action represents loading of the certain user's blog.
4. **User's blog with tag**
   Similarly to homepage with tag, this action represents loading of the user's blog with a specified tag. It contains only those user's posts, which have this tag assigned to their content.
5. **Post detail page**
   This action represents loading of the page with a concrete post.

6. **Insert comment page**
   This action represents loading of the page, where the user can insert a comment to a specified post or comment.

7. **Insert comment to post – execution**
   This action represents an insertion of a comment to a specified post.

8. **Insert comment to comment – execution**
   This action represents an insertion of a comment to another comment in a specified post.

9. **User's posts page**
   This action represents loading of the page, where user can manage his/her posts.

10. **Insert post page**
    This action represents loading of the page, where user can insert a post into his/her blog.

11. **Insert post – execution**
    This action represents an insertion of a post into user's blog.

12. **Update post page**
    This action represents loading of the page, where user can update a certain post in his/her blog.

13. **Update post – execution**
    This action represents updating a certain post in user's blog.

When the benchmark executes the certain action, it executes a respective group of statements in that order, in which they are executed on the portal during the real user's request. Statement characteristics of actions are in Table 7.2. As we see, some of the actions are read-only (they contain `select` statements solely), others modify the database in some way. If a table cell contains also a number in brackets, this number is valid for the action located in Optimized actions set. Otherwise, actions of both sets share the same feature. The benchmark can simulate users, which are logged or not logged in. In case, users are logged in, each action in both sets contains 4 additional `select` statements, which retrieve personal data of logged in user. These statements are executed firstly before other action statements.

Action name	**Statements in total**	**Select statements**	**Modification statements**
Homepage	4	4	0
Homepage with tag	4	4	0
User's blog	6	6	0
User's blog with tag	6	6	0
Post detail page	8	7	1
Insert comment page	3	3	0
Insert comment to post – execution	10 (11)	8	2 (3)
Insert comment to comment – execution	10 (11)	8	2 (3)
User's posts page	2	2	0
Insert post page	1	1	0
Insert post – execution	4 (5)	3	1 (2)
Update post page	2	2	0
Update post – execution	3	2	1

Table 7.2: Characteristics of action statements. This table contains the information about which statement types are used in actions.

Rules for executing the workload

In this section we specify, how the benchmark is used and how we make the measurements.

When running the benchmark, four options can be set – `USERS_NUM`, `LOGGED`, `INS_RATIO` and `OPT`. The first option defines how many users are going to be simulated. The second option defines, if these users are logged or not logged in. The third option specifies the ratio of users, who insert a post/comment, to users, who do not insert anything. These three options are required. The fourth option is not required and if it is set, Optimized actions set is used during the benchmarking. Otherwise Original actions set is used.

When the benchmark starts, it runs defined number of users parallelly. Each user is simulated by a PHP script, which executes above mentioned actions. In the beginning, values, which replace the keywords used in statements, are selected. The set of possible values is stored in two XML files. The first file contains values replacing first four keywords from the keyword list. These values specify a user, who's blog is viewed by other user. In the file, there are

User type	Permitted actions
Read-only, not logged in	1,2,3,4,5
Insert, not logged in	1,2,3,4,5,6,7,8
Read-only, logged in	1,2,3,4,5
Insert, logged in	all

Table 7.3: Types of users simulated in the benchmark and their permitted actions

50 users. The second file contains values replacing last three keywords from the keyword list. These values specify a user, who is logged in at the portal. In the file, there are all portal registered users – 1075. Which values are selected depends on a user's serial number and a total number of simulated users. This values selection guarantees that a user with a specific serial number executes the same versions of statements during each benchmark running. This helps us to compare the optimization contribution better.

When values replacing the keywords are selected, actions start to execute. Execution of an action means, that statements with replaced keywords are executed in the defined order. After execution of all statements belonging to a certain action, their execution times are count together and this number creates an execution time of the whole action. The order of executed actions is fixed for all users, but which actions are executed, depends on a type of the user, who is simulated. Table 7.3 shows the permitted actions for specific user types. The user, who is not logged in, cannot execute the last five actions from the action list, because he/she does not have a blog. The logged in user may execute each of the mentioned actions. The user marked as a read-only cannot insert a post/comment, otherwise he/she can insert a post/comment (post only, if the user is logged in). To simulate the user's behaviour the greatest extent possible, we add a pause between executed actions, which lasts from 5 to 9 seconds (precise number is computed from user's serial number). This can simulate the situation, when the user views some page and does not click on other link immediately, but waits and reads information displayed on the page.

To simulate the different contribution of each optimization type (statement

Option setting no.	USERS_NUM	LOGGED	INS_RATIO
1.	20	0	1:5
2.	20	1	1:5
3.	40	0	1:10
4.	40	1	1:10

Table 7.4: Different option settings used during the benchmarking

optimization, session and settings optimization), we run this benchmark five times with different optimization techniques used and compared the results. Below is the list of run benchmarks with their characteristics.

1. **Original statements benchmark**
   This benchmark uses Original actions set and all original Sedna settings.
2. **Original statements + settings benchmark**
   This benchmark uses Original actions set, but additionally to 1. benchmark, it uses optimized Sedna system settings (as proposed in Section 4.4.1).
3. **Original statements + read-only mode benchmark**
   This benchmark uses Original actions set, but additionally to 1. benchmark, it uses read-only mode with `select` statements (as proposed in Section 4.4.2).
4. **Optimized statements benchmark**
   Contrary to 1. benchmark, this benchmark uses Optimized actions set and all original Sedna settings.
5. **Optimized statements + settings + read-only mode benchmark**
   This benchmark uses Optimized actions set, optimized Sedna system settings and read-only mode with `select` statements – all optimization techniques together.

Each of this benchmarks was run with four different settings of the benchmark options (see Table 7.4) and each such benchmark was iterated five times for more precise computation of an average execution time of actions. In total, we run 100 benchmarks.

Metrics

The only one metric used in Complex benchmark is an average action execution time measured in seconds.

How to interpret the results

Since the benchmark metric is an average action execution time, the interpretation of the results is based on a comparison the average execution time of actions executed in benchmarks using different optimization techniques. Those optimization techniques, which boosted the average action execution time the most, are marked as the best optimization solution.

7.3.2 The results

As we said earlier, we run five benchmarks with different optimization techniques used. Each of these benchmarks was executed with different option settings (see Table 7.4). The benchmark results are on Figures 7.2, 7.3, 7.4 and 7.5. Each of these figures shows, how long the listed actions were executed (measured in seconds) during the benchmarking with different option settings and with different optimization techniques used. We can easily see on these figures, what the average execution time of actions with original statements was (1. benchmark), how the individual optimization techniques affected the average action execution time (benchmarks 2. – 4.) and what impact on the action execution time all these optimization techniques had together. Actions and benchmarks in the figures are numbered according to their lists mentioned in Section 7.3.1.

As we see on the figures, actions executed during Original statements benchmark (no. 1) did not have the worst average execution time always. There were cases, when some actions had their worst average execution time generated by Original statements + settings benchmark (2. benchmark), i.e., actions no. 7., 8., 10., 12., 13. on Figure 7.3. In our opinion, this was caused by increased page swapping. The optimization of the Sedna system settings resided, i.a., in increase of the number of buffers used by the Sedna database manager in main memory from 1600 (100 MB) to 8000 (500 MB) (as said in

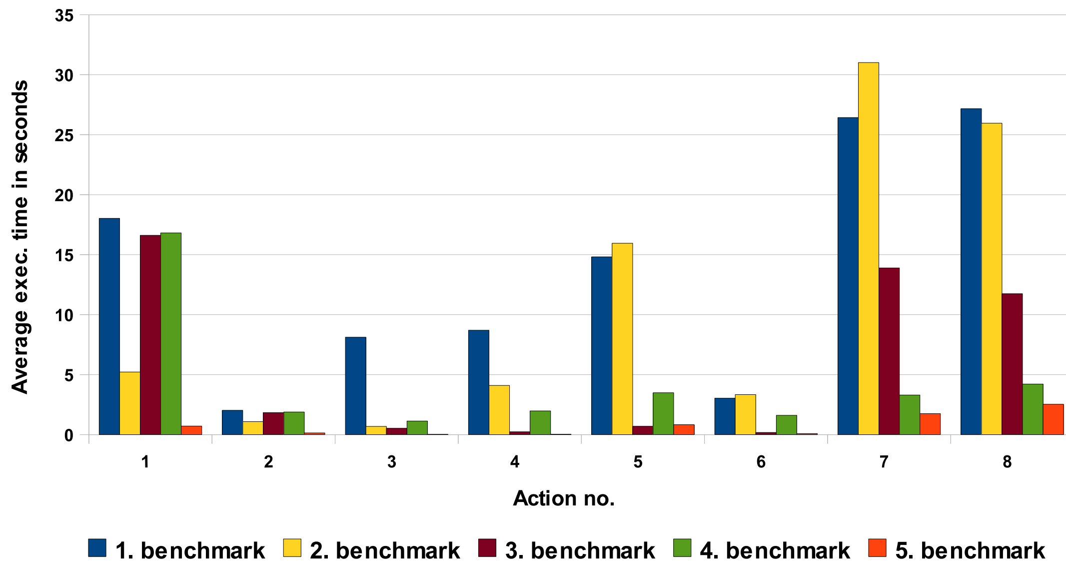


Figure 7.2: Complex benchmark results. Benchmark run with 20 not logged in users, every fifth user inserted a comment to a post/comment.

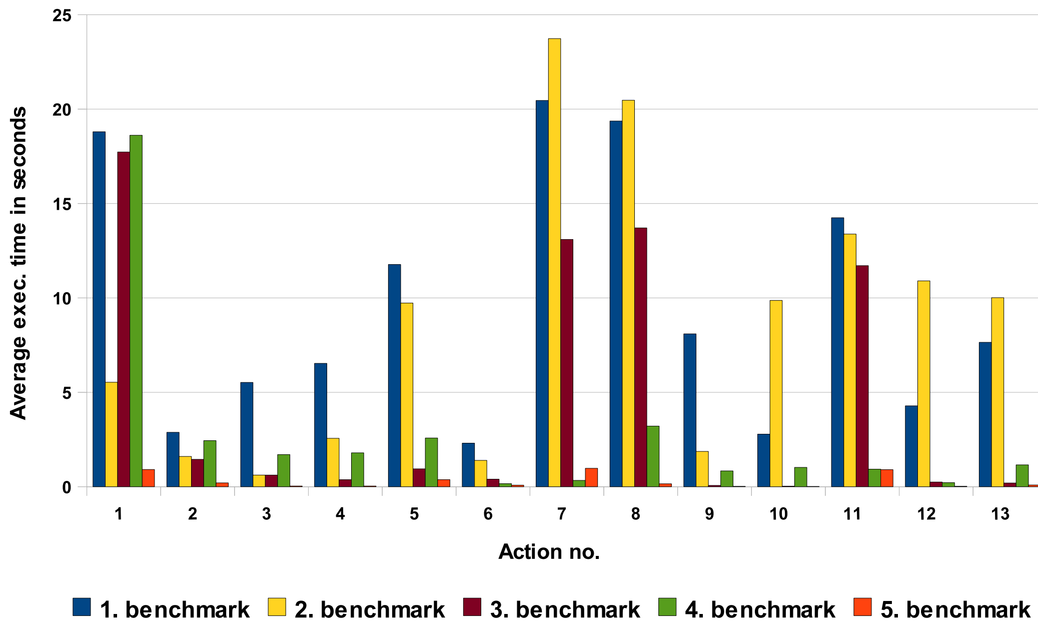


Figure 7.3: Complex benchmark results. Benchmark run with 20 logged in users, every fifth user inserted a comment or a post.

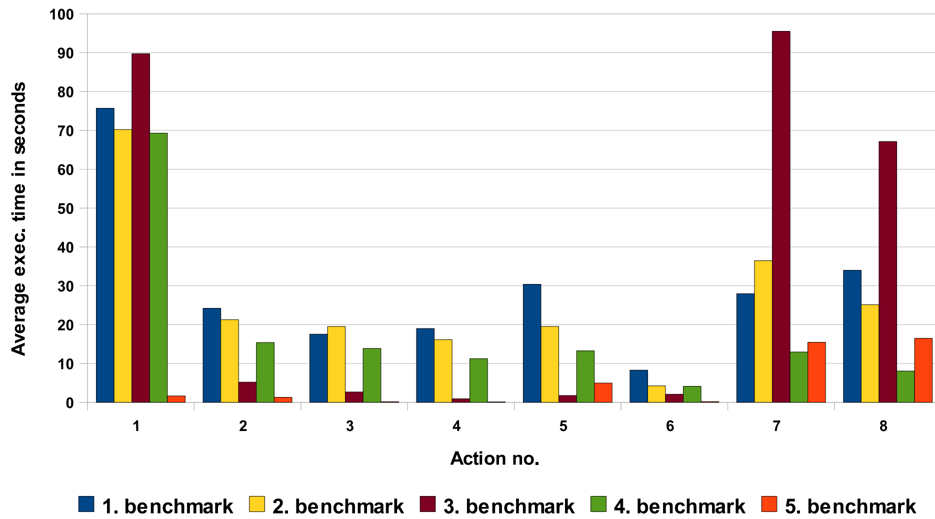


Figure 7.4: Complex benchmark results. Benchmark run with 40 not logged in users, every tenth user inserted a comment to a post/comment.

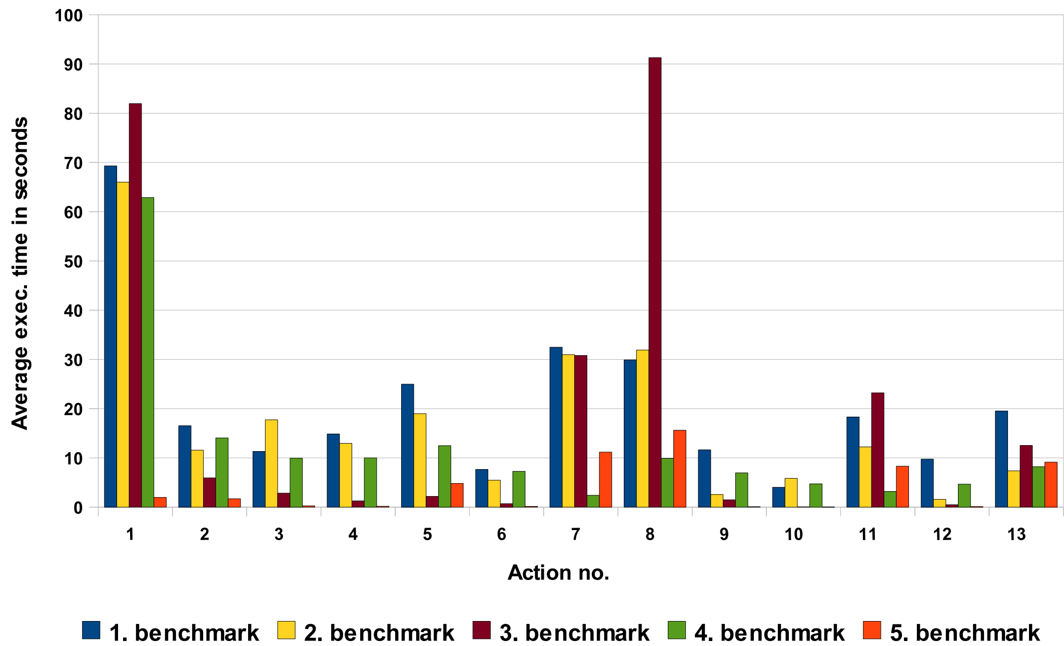


Figure 7.5: Complex benchmark results. Benchmark run with 40 logged in users, every tenth user inserted a comment or a post.

Section 4.4.1). This reduced the amount of main memory, which could be used by other Sedna processes (i.e., session processes) and increased page swapping. This caused an increase of an average execution time of actions compared to an average execution time of actions generated by Original statements benchmark.

Other case, where the worst average action execution time was generated by other than Original statements benchmark, was Original statements + read-only mode benchmark (3. benchmark) during the simulation of 40 users. Examples are actions no. 1., 7., 8. on Figure 7.4, which had their worst times generated by this benchmark. As we see, during the simulation of 20 users, the optimization technique of this benchmark (read-only mode) showed as a good solution for the database performance boost. But during the simulation of 40 users, the positive effect of this optimization technique turned into drawback in actions, which statements modified XML documents (no. 7., 8.) or passed through the whole users' collection more times (like the action no. 1 simulating the homepage loading).

Optimized statements benchmark (4. benchmark), unlike previous mentioned benchmarks, generated better average action execution time than Original statements benchmark each time except one case (action no. 10 on Figure 7.5, where 40 logged in users were simulated). So this optimization technique showed as the best individual optimization technique used during the benchmarking.

Last benchmark, which results we want to describe, is Optimized statements + settings + read-only mode benchmark (5. benchmark). This benchmark used all optimization techniques together. The result was that in majority of cases it generated better average action execution time than 4. benchmark, as we assumed. But in cases, where benchmarks no. 2. or 3. generated worse average execution time of actions than 1. benchmark, this influenced the results of this benchmark and the average execution time of certain actions was longer than in 4. benchmark. Nevertheless, the overall positive contribution of all optimization techniques used in 5. benchmark was a lot greater than positive contribution of individual optimization techniques, since our portal is loaded mainly by the `select` statements, as we declared in Section 6.2.1.

Optimization techniques used together in 5. benchmark boosted the performance of every listed action significantly. The biggest success for us is the performance boost of the homepage loading and insertion of the comment/-post, since these actions were weak points of our portal. The homepage average execution time boosted more than 95% in total in all four benchmark option settings. Insertion of the comment/post boosted more than 90% in first two benchmark option settings and more than 40% in last two benchmark option settings.

Conclusion

Our work can be divided into three main parts. In the first part we introduced the basic building blocks of our topic – XML format, XML databases and Sedna database management system deployed at the portal. The second part contained the analysis of the existing optimization of the database, optimization proposals and description of existing benchmarks. The third part described the analysis of database statements, the desing of our benchmarks and their results.

During the introduction of the basic building blocks, we described in detail, how the XML format looks like. We described its structural components and languages, which are used for its transformation and querying. The second chapter presented XML databases and their main features. We explained, for what they can be used for and why they should be used. We introduced the basic segmentation of XML databases and described in detail native XML databases, on which our thesis is based on. The third chapter was dedicated to Sedna, the XML database deployed at the blog.matfyz.sk portal. We described their architecture, storage system, update language and basic features of the transactions used in this system.

The main function of the second part of our work was to introduce our optimization proposals and make the analysis of existing benchmarks. Firstly, we analysed the hardware and software used on our server. Then we evaluated existing database optimization and proposed optimization techniques, which could be used during the database optimization. Next, the existing benchmarks were described. After their description, we proposed to create our own benchmarks, since no one from mentioned benchmarks satisfied our conditions.

Third part consisted of the database statement analysis and portal benchmarks design with presentation of their results. The statement analysis was made, because we did not have information about which statements are used on the portal and how long they last. After this analysis we knew, that the database was loaded mainly by reading the data, not by their modification. We also knew precisely, what statements caused the biggest problem during their execution. Based on this analysis we proposed the appropriate index set, which was then used during the benchmarking.

We created two benchmarks. The first benchmark measured, how the statements optimization proposed in the second part of work boosted their original average execution time. The results showed great performance improvement. The second benchmark used these optimized statements during benchmarking the next aspect of the database – how the performance of the whole actions executed on the portal was improved by optimization techniques proposed in the second part of the work. This benchmark used multi-user environment – it simulated multiple users in parallel. We showed that our proposed optimization techniques boosted the performance of all actions executed on the portal significantly. For example, the average execution time of the homepage loading action boosted more than 95%. The actions representing an insertion of the post or the comment, which were among the actions with the longest average execution time, boosted more than 40% and in some case more than 90%.

Bibliography

[ABNE10] Mohammed Al-Badawi, Siobhán North, and Barry Eaglestone. The 3D XML Benchmark. In *WEBIST '09: 6th International Conference on Web Information Systems and Technologies*, April 2010.

[AMM05] Loredana Afanasiev, Ioana Manolescu, and Philippe Michiels. MemBeR: A Micro-benchmark Repository for XQuery. In Stéphane Bressan, Stefano Ceri, Ela Hunt, Zachary G. Ives, Zohra Bellahsene, Michael Rys, and Rainer Unland, editors, *XSym*, volume 3671 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2005.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[BMY09] Denilson Barbosa, Ioana Manolescu, and Jeffrey Xu Yu. XML Benchmarks. In *Encyclopedia of Database Systems*, pages 3576–3579. Springer US, 2009.

[Bou05] Ronald Bourret. XML and Databases, September 2005. Web article `http://www.rpbourret.com/xml/XMLAndDatabases.htm`.

[BR02] Timo Böhme and Erhard Rahm. Multi-user Evaluation of XML Data Management Systems with XMach-1. In *In Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Tech-*

niques and Data Integration over the Web-Revised Papers, pages 167–174. Springer-Verlag, 2002.

[Cor09] Oracle Corporation. XQuery API for Java, 2009. http://jcp.org/en/jsr/detail?id=225.

[fSPR] Institute for System Programming RAS. Sedna XML Database Download Page. http://www.sedna.org/download.html.

[fSPR94] Institute for System Programming RAS. About isp ras, January 1994. http://www.ispras.ru/en/index.php.

[fSPR10] Institute for System Programming RAS. PHP API for Sedna XML database, September 2010. http://www.sedna.org/download.html.

[fSPR11a] Institute for System Programming RAS. Sedna Administration Guide, October 2011. http://www.sedna.org/adminguide/AdminGuide.html.

[fSPR11b] Institute for System Programming RAS. Sedna Programmer's Guide, October 2011. http://www.sedna.org/progguide/ProgGuide.html.

[Gro] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/docs/8.2/static/datatype-xml.html.

[Inf] 3D Informatica. ExtraWay XML Engine. Native XML database http://www.3di.it/en/products/tecnologia/extraway.

[Ini03] The XML:DB Initiative. XML databases, 2003. http://xmldb-org.sourceforge.net/.

[Koh08] Anton Kohutovič. blog.matfyz.sk community blog portal. Master thesis, Comenius university, 2008.

[Leh01] Patrick Lehti. Design and Implementation of a Data Manipulation Processor for an XML Query Language. Master thesis, Technische Universität Darmstadt, 2001.

[lWYsC93] Kun lung Wu, Philip S. Yu, and Ming syan Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In *In Proceedings of the Ninth International Conference on Data Engineering*, pages 577–586, 1993.

[McO] McObject. eXtremeDB – real-time embedded database. `http://www.mcobject.com/extremedbfamily.shtml`.

[Mei00] Wolfgang Meier. eXist-db Open Source Native XML Database, 2000. Native XML database `http://exist-db.org/`.

[Mly08] Irena Mlynkova. XML Benchmarking. In *Proceedings of the IADIS Multi Conference on Computer Science and Information Systems - subconference Informatics 2008*, MCCSIS '08, pages 59–66. IADIS, July 2008.

[NKS07] Matthias Nicola, Irina Kogan, and Berni Schiefer. An XML Transaction Processing Benchmark. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 937–948, New York, NY, USA, 2007. ACM.

[Ozo] Ozone. Ozone - Java OODBMS. Native XML database `http://sourceforge.net/projects/ozone/`.

[Rej10] Martin Rejda. Modular Redesign of the blog.matfyz.sk Portal. Master thesis, Comenius university, 2010.

[RPJ+06] Kanda Runapongsa, Jignesh M. Patel, H. V. Jagadish, Yun Chen, and Shurug Al-Khalifa. The Michigan benchmark: towards XML query performance diagnostics. *Inf. Syst.*, 31(2):73–97, April 2006.

[SG01] Aaron Skonnard and Martin Gudgin. *Essential Xml Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2001.

[SKS01] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill Higher Education, 4th edition, 2001.

[Tea05] BaseX Team. BaseX. The XML Database., 2005. Native XML database http://basex.org/.

[Tec] Orient Technologies. Orient ODBMS. http://www.orientechnologies.com/cms/?Solutions:Orient_ODBMS.

[TSK+10] Ilya Taranov, Ivan Shcheklein, Alexander Kalinin, Leonid Novak, Sergei Kuznetsov, Roman Pastukhov, Alexander Boldakov, Denis Turdakov, Konstantin Antipin, Andrey Fomichev, Peter Pleshachkov, Pavel Velikhov, Nikolai Zavaritski, Maxim Grinev, Maria Grineva, and Dmitry Lizorkin. Sedna: native XML database management system (internals overview). In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1037–1046, New York, NY, USA, 2010. ACM.

[Ďu11] Juraj Ďuďák. Tagging System for the blog.matfyz.sk Portal. Master thesis, Comenius university, 2011.

[W3C05] W3C. Document Object Model (DOM), January 2005. http://www.w3.org/DOM/.

[W3C07] W3C. XSL Transformations (XSLT) Version 2.0, January 2007. W3C Recommendation http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[W3C08] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008. W3C Recommendation http://www.w3.org/TR/2008/REC-xml-20081126/.

[W3C09] W3C. Namespaces in XML 1.0 (Third Edition), December 2009. W3C Recommendation http://www.w3.org/TR/2009/REC-xml-names-20091208/.

[W3C10a] W3C. XML Linking Language (XLink) Version 1.1, May 2010. W3C Recommendation http://www.w3.org/TR/2010/REC-xlink11-20100506/.

[W3C10b] W3C. XML Path Language (XPath) 2.0 (Second Edition), December 2010. W3C Recommendation `http://www.w3.org/TR/2010/REC-xpath20-20101214/`.

[W3C10c] W3C. XQuery 1.0: An XML Query Language (Second Edition), December 2010. W3C Recommendation `http://www.w3.org/TR/2010/REC-xquery-20101214/`.

[W3C10d] W3C. XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition), December 2010. W3C Recommendation `http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/`.

[W3C10e] W3C. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition), December 2010. W3C Recommendation `http://www.w3.org/TR/2010/REC-xpath-functions-20101214/`.

[W3C11a] W3C. XQuery 3.0: An XML Query Language, June 2011. W3C Working Draft `http://www.w3.org/TR/2011/WD-xquery-30-20110614/`.

[W3C11b] W3C. XQuery Update Facility 1.0, March 2011. W3C Recommendation `http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/`.

[Wal07] Walmsley, Priscilla. *XQuery*. O'Reilly Media, Inc., 2007.

[YOK04] Benjamin Bin Yao, M. Tamer Özsu, and Nitin Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 621–632, Washington, DC, USA, 2004. IEEE Computer Society.

Appendix A

Simple benchmark

A.1 Operation sets

A.1.1 Original statements set

```
1. statement (Get user's ID):
  collection("weblog")/user[info/nick="###USER_NICK###"]/@ID/normalize-space()
```

```
2. statement (Get 20 user's posts with certain tag ordered by timestamp):
declare option se:output "indent=no";
  let $all := collection("weblog")/user/blog/post[@status="published"][
      contains("sk",@lang)][@ID= ("p16085","p16090","p16095","p16116","p13362
      ","p15957","p15967","p15991","p16054","p16057")][../../@ID="###USER_ID
      ###"],
  $list := subsequence(
        for $i in $all
        order by (xs:integer($i/date/@timestamp))
        descending return $i,
    1,20),
  $last := count($all),
  $posts:=
  (for $current in $list
  let $user := $current/ancestor::user
  return
  <post>{$current/@*}
    <user>{$user/@ID}{$user/@type}
    {$user/info/nick}
    {$user/info/realName}
    </user>
    {$current/*}
  </post>)
```

```
  return <module name="Articles"><newPosts last="{$last}" end="20" start="1">{
      $posts}</newPosts></module>
```

```
3. statement (Get user's blog title):
  data(collection("weblog")/user[@ID="###USER_ID###"]/blog/title)
```

```
4. statement (Get user's info):
  collection("weblog")/user[@ID="###USER_ID###"]/info
```

```
5. statement (Get blog customization):
  let $bg := collection("weblog")/user[@ID="###USER_ID###"]/blog/background,
    $color := collection("weblog")/user[@ID="###USER_ID###"]/blog/colorscheme
    return
   <customizations>
     <blogBackground>{$bg}</blogBackground>
     <colorScheme>{$color}</colorScheme>
   </customizations>
```

```
6. statement (Get the newest comment from user's posts):
  let $all := collection("weblog")/user/blog/post[@status="published"][
      contains("sk",@lang)][@ID= ("p16085","p16090","p16095","p16116","p13362
      ","p15957","p15967","p15991","p16054","p16057")][../../@ID="###USER_ID
      ###"]//comment,
  $last := subsequence(
          for $i in $all
          order by (xs:integer($i/date/@timestamp))
          ascending return $i,
        count($all),1)
      return
      <post>{$last/ancestor::post/@*}
        {$last/ancestor::post/title}
        {$last/ancestor::post/subtitle}
        {$last}
      </post>
```

```
7. statement (Get certain post's title text):
  for $post in collection("weblog")/user/blog/post[@ID="###POST_ID###"] return
       $post/title/text()
```

```
8. statement (Get post's user's nick):
  for $post in collection("weblog")/user/blog/post[@ID="###POST_ID###"] return
       $post/../../info/nick/text()
```

```
9. statement (Get user's type):
  data(collection("weblog")/user[@ID="###USER_ID###"]/@type)
```

```
10. statement (Get all user's posts except one):
  for $i in collection("weblog")/user/blog/post[@status="published"][../../@ID
      ="###USER_ID###"][@ID!="###POST_ID###"]
        order by (xs:integer($i/date/@timestamp))
        descending
```

```
return
<post>
  {$i/@status}
  {$i/@private}
  {$i/@ID}
  {$i/title}
  {$i/date}
</post>
```

```
11. statement (Get post accessCount):
  data(collection("weblog")/user[@ID="###USER_ID###"]/blog/post[@ID="###
      POST_ID###"]/@accessCount)
```

```
12. statement (Update accessCount in certain post):
  update replace $i in collection("weblog")/user[@ID="###USER_ID###"]/blog/
      post[@ID="###POST_ID###"]/@accessCount with attribute accessCount {1033}
```

```
13. statement (Get user type in course):
  data(collection("course")/courses/course[@year="2011/2012"]/users/user[@ref
      ="###USER_NICK###"]/@type)
```

```
14. statement (Get all user's posts):
  declare option se:output "indent=no";
      let $info := collection("weblog")/user[@ID="###USER_ID###"]/info
      let $all := collection("weblog")/user/blog/post[../../@ID="###USER_ID
          ###"],
      $list := subsequence(for $i in $all return $i,1,count($all)),
      $user := (
        <info>
          {$info/nick}
          {$info/realName}
        </info> ),
      $posts:= ( for $current in $list
       return
         <post>
           {$current/@*}
           {$current/*}
         </post>
      )
      return
        <user>
          {$user}
          <blog>
            {$info/../blog/title}
            {$posts}
          </blog>
        </user>
```

```
15. statement (Get users with at least one slovak post):
  for $i in collection("weblog")/user
    let $lastName := if ($i/info/lastName!="") then lower-case(translate($i/
```

```
        info/lastName/text(), "aeiouycdllnorstzaeiouycdllnorstz", "
        aeiouycdllnorstzaeiouycdllnorstz"))
      else $i/info/nick/text(),
    $realName:=if ($i/info/realName!="") then $i/info/realName/text() else $i/
        info/nick/text()
  where count($i/blog/post[contains("sk",@lang)])>0
  order by $lastName ascending
  return
      <user>
    {$i/info/nick}
    {$i/info/ranks}
    {$i/info/email}
    <realName>{$realName}</realName>
    <lastName>{$lastName}</lastName>
      </user>
----------------------------------------------------------------
16. statement (Get 10 best ranked published slovak posts):
  subsequence(
    (for $i in collection("weblog")/user/blog/post[@status="published"][
        contains("sk",@lang)][ranks/reputationScore>5]
    let $user := $i/ancestor::user
    order by (
      if ($i/ranks) then xs:double($i/ranks/reputationScore)
      else xs:double(0.0)
    )
    descending
    return
    <post>{$i/@*}
      <user>{$user/@ID}{$user/@type} {$user/info/nick} {$user/info/realName}
      </user>
    {$i/title}
    {$i/subtitle}
    {$i/ranks}
    </post>)
  ,1,10)
----------------------------------------------------------------
17. statement (Get 10 most read published slovak posts with access higher than
     50):
  subsequence(
    (for $i in collection("weblog")/user/blog/post[@status="published"][
        contains("sk",@lang)][@accessCount>50]
    let $user := $i/ancestor::user
    order by xs:integer($i/@accessCount) descending
    return
    <post>{$i/@ID}{$i/@lang}{$i/@accessCount}
    <user>{$user/@ID}{$user/@type} {$user/info/nick} {$user/info/realName}
    </user>
    {$i/title}
    {$i/subtitle}
    </post>)
  ,1,10)
```

```
18. statement (Create upload form for user):
  let $user:=collection("weblog")/user[@ID="###USER_ID###"]
  return
    <uploadForm> {$user/blog/@css}
    <userID></userID>
    {$user/info/*}
    {$user/blog/title}
    <userFiles size="16.04 kB"><file size="10.96 kB">avatar.jpg</file><file
        size="1.57 kB">blog.xsl</file><file size="3.52 kB">header.jpg</file></
        userFiles>
    </uploadForm>
```

```
19. statement (Get post votes count):
  collection("weblog")/user[@ID=###USER_ID###]/blog/post[@ID="###POST_ID###"]/
      votes/@count
```

```
20. statement (Insert votes tag to post):
  update insert <votes count='1'></votes> into collection('weblog')/user[@ID
      =###USER_ID###]/blog/post[@ID='###POST_ID###']
```

```
21. statement (Get password text):
  collection("weblog")/user[info/nick="###USER_NICK###"]/account/password/text
      ()
```

```
22. statement (Check if comment with certain ID exists):
  for $i in collection("weblog")/user/blog/post//comment[@ID="###COMMENT_ID
      ###"] return 1
```

```
23. statement (Insert comment to comment):
  update insert
<comment lang="en" ref="p991" ID="###MAX_ID###" type="xhtml"><title>indeed</
    title><content><p>... you are right, I should have</p></content><date
    timestamp="1319478009">2011-10-24T19:40:09</date><author userID="u968"><
    nick>mirax33</nick><realName>Miroslav Pot</realName></author></comment>
 into collection("weblog")/user/blog/post//comment[@ID="###COMMENT_ID###"]
```

```
24. statement (Insert post to blog):
  declare option se:output 'indent=no'; update insert
    <post lang="en" type="xhtml" status="published" private="no" accessCount
        ="0" ID="###MAX_ID###">
      <title>Hello world!</title>
      <subtitle/>
      <content>
        <p> Then the day ... (all content available on enclosed CD) ... </p>
      </content>
      <tags>
        <tag>story</tag>
      </tags>
    <date timestamp="1319436071">2011-10-24T08:01:11</date><topics></topics><
        comments></comments></post>
```

```
    into collection('weblog')/user[@ID="###USER_ID###"]/blog
```

```
25. statement (Update post):
  declare option se:output "indent=no"; update replace $i in collection("
      weblog")/user[@ID="###USER_ID###"]/blog/post[@ID="###POST_ID###"] with
<post lang="en" type="xhtml" status="published" accessCount="1" private="no"
    ID="###POST_ID###" submitted="yes">
  <title>2nd chapter &#x2013; Stealing doesn't pay off.</title>
  <subtitle/>
  <content>
    <p> Then the day ... (all content available on enclosed CD) ... </p>
  </content>
  <tags>
    <tag>story</tag>
  </tags>
  <date timestamp="1319436071">2011-10-24T08:01:11</date>
  <topics/>
  <comments/>
<lastUpdate timestamp="1319437105">2011-10-24T08:18:25</lastUpdate></post>
```

```
26. statement (Check if post with certain ID exists):
  for $i in collection("weblog")/user/blog/post[@ID="###POST_ID###"] return 1
```

```
27. statement (Get max used ID):
  max(collection('weblog')/user/blog//*[name()='post' or name()='comment']/
      number(substring-after(@ID,'p')))
```

A.1.2 Optimized statements set

```
2. statement (Get 20 user's posts ordered by timestamp):
  declare option se:output "indent=no";
  let $all := collection("weblog")/user[@ID="###USER_ID###"]/blog/post[@status
      ="published"][contains("sk",@lang)][@ID= ("p16085","p16090","p16095","
      p16116","p13362","p15957","p15967","p15991","p16054","p16057")],
  $list := subsequence(for $i in $all
                          order by (xs:integer($i/date/@timestamp)) descending
                          return $i,
           1,20),
  $last := count($all),
  $posts:=
    (for $current in $list
    let $user := $current/ancestor::user
    return
    <post>{$current/@*}
      <user>{$user/@ID}{$user/@type}
      {$user/info/nick}
      {$user/info/realName}
      </user>
      {$current/*}
    </post>
```

```
    )
  return <module name="Articles"><newPosts last="{$last}" end="20" start="1">{
      $posts}</newPosts></module>
```

```
5. statement (Get blog customization):
  let $userBlog := collection("weblog")/user[@ID="###USER_ID###"]/blog
  return <customizations> <blogBackground>{$userBlog/background}</
      blogBackground> <colorScheme>{$userBlog/colorscheme}</colorScheme> </
      customizations>
```

```
6. statement (Get the newest comment from user's posts):
  let $all := collection("weblog")/user[@ID="###USER_ID###"]/blog/post[@status
      ="published"][contains("sk",@lang)][@ID= ("p16085","p16090","p16095","
      p16116","p13362","p15957","p15967","p15991","p16054","p16057")]//comment
      ,
  $last :=  subsequence(
          for $i in $all
          order by (xs:integer($i/date/@timestamp)) ascending
          return $i,count($all),1)
  return
  <post>{$last/ancestor::post/@*}
    {$last/ancestor::post/title}
    {$last/ancestor::post/subtitle}
    {$last}
  </post>
```

```
7. statement (Get certain post's title text):
  collection("weblog")/user/blog/post[@ID="###POST_ID###"]/title/text()
```

```
8. statement (Get post's user's nick):
  collection("weblog")/user/blog/post[@ID="###POST_ID###"]/../../info/nick/
      text()
```

```
10. statement (Get all user's posts except one):
  for $i in collection("weblog")/user[@ID="###USER_ID###"]/blog/post[@status="
      published"][@ID!="###POST_ID###"]
  order by (xs:integer($i/date/@timestamp)) descending
  return
  <post>
    {$i/@status}
    {$i/@private}
    {$i/@ID}
    {$i/title}
    {$i/date}
  </post>
```

```
14. statement (Get all user's posts):
  declare option se:output "indent=no";
  let $all := collection("weblog")/user[@ID="###USER_ID###"]/blog/post,
  $user := (
      <info>
```

```
          {$all/../../info/nick}
          {$all/../../info/realName}
      </info> ),
  $posts:= ( for $current in $all
   return
     <post>
       {$current/@*}
       {$current/*}
     </post>
  )
  return
    <user>
  {$user}
  <blog>
      {$all/../title}
      {$posts}
  </blog>
  </user>
```

```
15. statement (Get users with at least one slovak post):
  for $i in collection("weblog")/user[count(blog/post[contains("sk",@lang)])
      >0]
  let $lastName:=if ($i/info/lastName!="") then lower-case(translate($i/info/
      lastName/text(),"aeiouycdllnorstzaeiouycdllnorstz","
      aeiouycdllnorstzaeiouycdllnorstz"))
      else $i/info/nick/text(),
  $realName:=if ($i/info/realName!="") then $i/info/realName/text() else $i/
      info/nick/text()
  order by $lastName ascending
  return
  <user>
            {$i/info/nick}
            {$i/info/ranks}
            {$i/info/email}
            <realName>{$realName}</realName>
            <lastName>{$lastName}</lastName>
  </user>
```

```
22. statement (Check if comment with certain ID exists):
  if (collection("weblog")/user/blog/post//comment[@ID="###COMMENT_ID###"])
      then 1 else 0
```

```
26. statement (Check if post with certain ID exists):
  if (collection("weblog")/user/blog/post[@ID="###POST_ID###"]) then 1 else 0
```

```
27. statement (Get max used ID):
  data(collection("maximumId")/maximumId/postComment)
```

```
28. statement (Get max used ID):
  update replace $i in collection("maximumId")/maximumId/postComment with <
      postComment>###MAX_ID###</postComment>
```

A.1.3 Index statements set

```
1. statement (Get user's ID):
  index-scan("userNick","###USER_NICK###","EQ")/@ID/normalize-space()
```

```
2. statement (Get 20 user's posts ordered by timestamp):
  declare option se:output "indent=no";
  let $all := index-scan("userID","###USER_ID###","EQ")/blog/post[@status="
      published"][contains("sk",@lang)][@ID= ("p16085","p16090","p16095","
      p16116","p13362","p15957","p15967","p15991","p16054","p16057")],
  $list := subsequence(for $i in $all
                          order by (xs:integer($i/date/@timestamp)) descending
                          return $i,
          1,20),
  $last := count($all),
  $posts:=
  (for $current in $list
  let $user := $current/ancestor::user
  return
  <post>{$current/@*}
    <user>{$user/@ID}{$user/@type}
    {$user/info/nick}
    {$user/info/realName}
    </user>
    {$current/*}
  </post>
  )
  return <module name="Articles"><newPosts last="{$last}" end="20" start="1">{
      $posts}</newPosts></module>
```

```
3. statement (Get user's blog title):
  data(index-scan("userID","###USER_ID###","EQ")/blog/title)
```

```
4. statement (Get user's info):
  index-scan("userID","###USER_ID###","EQ")/info
```

```
5. statement (Get blog customization):
  let $userBlog := index-scan("userID","###USER_ID###","EQ")/blog
  return <customizations> <blogBackground>{$userBlog/background}</
      blogBackground> <colorScheme>{$userBlog/colorscheme}</colorScheme> </
      customizations>
```

```
6. statement (Get the newest comment from user's posts):
  let $all := index-scan("userID","###USER_ID###","EQ")/blog/post[@status="
      published"][contains("sk",@lang)][@ID= ("p16085","p16090","p16095","
      p16116","p13362","p15957","p15967","p15991","p16054","p16057")]//comment
      ,
  $last := subsequence(
          for $i in $all
          order by (xs:integer($i/date/@timestamp)) ascending
          return $i,count($all),1)
```

```
  return
  <post>{$last/ancestor::post/@*}
    {$last/ancestor::post/title}
    {$last/ancestor::post/subtitle}
    {$last}
  </post>
------------------------------------------------
7. statement (Get certain post's title text):
  index-scan("postID","###POST_ID###","EQ")/title/text()
------------------------------------------------
8. statement (Get post's user's nick):
  index-scan("postID","###POST_ID###","EQ")/../../info/nick/text()
------------------------------------------------
9. statement (Get user's type):
  data(index-scan("userID","###USER_ID###","EQ")/@type)
------------------------------------------------
10. statement (Get all user's posts except one):
  for $i in index-scan("userID","###USER_ID###","EQ")/blog/post[@status="
      published"][@ID!="###POST_ID###"]
  order by (
    xs:integer($i/date/@timestamp)
  )
  descending
  return
  <post>
    {$i/@status}
    {$i/@private}
    {$i/@ID}
    {$i/title}
    {$i/date}
  </post>
------------------------------------------------
11. statement (Get post accessCount):
  data(index-scan("postID","###POST_ID###","EQ")/@accessCount)
------------------------------------------------
12. statement (Update accessCount in certain post):
  update replace $i in index-scan("postID","###POST_ID###","EQ")/@accessCount
      with attribute accessCount {1033}
------------------------------------------------
13. statement (Get user type in course):
  data(index-scan("courseUserRef","###USER_NICK###","EQ")[../../@year
      ="2011/2012"]/@type)
------------------------------------------------
14. statement (Get all user's posts):
  declare option se:output "indent=no";
  let $all := index-scan("userID","###USER_ID###","EQ")/blog/post,
  $user := (
    <info>
        {$all/../../info/nick}
        {$all/../../info/realName}
    </info> ),
```

```
  $posts:= ( for $current in $all
   return
     <post>
       {$current/@*}
       {$current/*}
     </post>
  )
  return
    <user>
  {$user}
  <blog>
      {$all/../title}
      {$posts}
  </blog>
  </user>
```

```
16. statement (Get 10 best ranked published slovak posts):
  subsequence(
    (for $i in index-scan("postReputation","5","GT")[@status="published"][
        contains("sk",@lang)]
    let $user := $i/ancestor::user
    order by (
      if ($i/ranks) then xs:double($i/ranks/reputationScore)
      else xs:double(0.0)
    )
    descending
    return
    <post>{$i/@*}
      <user>{$user/@ID}{$user/@type} {$user/info/nick} {$user/info/realName}
      </user>
    {$i/title}
    {$i/subtitle}
    {$i/ranks}
    </post>)
  ,1,10)
```

```
17. statement (Get 10 most read published slovak posts with access higher than
     50):
  subsequence(
    (for $i in index-scan("postAccessCount","50","GT")[@status="published"][
        contains("sk",@lang)]
    let $user := $i/ancestor::user
    order by xs:integer($i/@accessCount) descending
    return
    <post>{$i/@ID}{$i/@lang}{$i/@accessCount}
    <user>{$user/@ID}{$user/@type} {$user/info/nick} {$user/info/realName}
    </user>
    {$i/title}
    {$i/subtitle}
    </post>)
  ,1,10)
```

```
18. statement (Create upload form for user):
  let $user:= index-scan("userID","###USER_ID###","EQ")
  return
    <uploadForm> {$user/blog/@css}
    <userID></userID>
    {$user/info/*}
    {$user/blog/title}
    <userFiles size="16.04 kB"><file size="10.96 kB">avatar.jpg</file><file
        size="1.57 kB">blog.xsl</file><file size="3.52 kB">header.jpg</file></
        userFiles>
    </uploadForm>
```

```
19. statement (Get post votes count):
  index-scan("postID","###POST_ID###","EQ")/votes/@count
```

```
20. statement (Insert votes tag to post):
  update insert <votes count='1'></votes> into index-scan("userID","###USER_ID
      ###","EQ")/blog/post[@ID='###POST_ID###'])
```

```
21. statement (Get password text):
  index-scan("userNick","###USER_NICK###","EQ")/account/password/text()
```

```
23. statement (Insert comment to comment):
  update insert
<comment lang="en" ref="p991" ID="###MAX_ID###" type="xhtml"><title>indeed</
    title><content><p>... you are right, I should have</p></content><date
    timestamp="1319478009">2011-10-24T19:40:09</date><author userID="u968"><
    nick>mirax33</nick><realName>Miroslav Pot</realName></author></comment>
 into index-scan("postID","###POST_ID###","EQ")//comment[@ID="###COMMENT_ID
     ###"
```

```
24. statement (Insert post to blog):
  declare option se:output 'indent=no'; update insert
  <post lang="en" type="xhtml" status="published" private="no" accessCount="0"
       ID="###MAX_ID###">
    <title>Hello world!</title>
    <subtitle/>
    <content>
      <p> Then the day of ... (all content available on enclosed CD) ... </p>
    </content>
    <tags>
      <tag>story</tag>
    </tags>
  <date timestamp="1319436071">2011-10-24T08:01:11</date><topics></topics><
      comments></comments></post>
  into index-scan("userID","###USER_ID###","EQ")/blog
```

```
25. statement (Update post):
  declare option se:output "indent=no"; update replace $i in index-scan("
      postID","###POST_ID###","EQ") with
```

```
<post lang="en" type="xhtml" status="published" accessCount="1" private="no"
    ID="###POST_ID###" submitted="yes">
  <title>2nd chapter &#x2013; Stealing doesn't pay off.</title>
  <subtitle/>
  <content>
    <p> Then the day of ... (all content available on enclosed CD) ... </p>
  </content>
  <tags>
    <tag>story</tag>
  </tags>
  <date timestamp="1319436071">2011-10-24T08:01:11</date>
  <topics/>
  <comments/>
<lastUpdate timestamp="1319437105">2011-10-24T08:18:25</lastUpdate></post>
```

```
26. statement (Check if post with certain ID exists):
  if (index-scan("postID","###POST_ID###","EQ")) then 1 else 0
```

A.1.4 Index II statements set

```
12. statement (Update accessCount in certain post):
  update replace $i in index-scan("postID","###POST_ID###","EQ")/@accessCount
      with attribute accessCount {1033}
```

```
17. statement (Get 10 most read published slovak posts with access higher than
     50):
  subsequence(
    (for $i in index-scan("postStatus","published","EQ")[contains("sk",@lang)
        ][@accessCount>50]
    let $user := $i/ancestor::user
    order by xs:integer($i/@accessCount) descending
    return
    <post>{$i/@ID}{$i/@lang}{$i/@accessCount}
    <user>{$user/@ID}{$user/@type} {$user/info/nick} {$user/info/realName}
    </user>
    {$i/title}
    {$i/subtitle}
    </post>)
  ,1,10)
```

```
24. statement (Insert post to blog):
  declare option se:output 'indent=no'; update insert
  <post lang="en" type="xhtml" status="published" private="no" accessCount="0"
      ID="###MAX_ID###">
    <title>Hello world!</title>
    <subtitle/>
    <content>
      <p> Then the day of ... (all content available on enclosed CD) ... </p>
    </content>
    <tags>
```

```
      <tag>story</tag>
    </tags>
  <date timestamp="1319436071">2011-10-24T08:01:11</date><topics></topics><
      comments></comments></post>
  into index-scan("userID","###USER_ID###","EQ")/blog
```

```
25. statement (Update post):
  declare option se:output "indent=no"; update replace $i in index-scan("
      postID","###POST_ID###","EQ") with
  <post lang="en" type="xhtml" status="published" accessCount="1" private="no"
       ID="###POST_ID###" submitted="yes">
    <title>2nd chapter &#x2013; Stealing doesn't pay off.</title>
    <subtitle/>
    <content>
      <p> Then the day of ... (all content available on enclosed CD) ... </p>
    </content>
    <tags>
      <tag>story</tag>
    </tags>
    <date timestamp="1319436071">2011-10-24T08:01:11</date>
    <topics/>
    <comments/>
  <lastUpdate timestamp="1319437105">2011-10-24T08:18:25</lastUpdate></post>
```

A.2 Partial results

Statement ID	Total Execution Time	Minimal Execution Time	Maximal Execution Time	Average Execution Time	Standard Deviation
1	2 085.19	2.03	6.59	4.17	1.21
2	18 597.91	36.45	55.26	37.20	1.26
3	1 728.97	0.91	15.12	3.46	1.27
4	2 259.84	1.80	6.53	4.52	0.96
5	4 808.12	9.43	27.92	9.62	0.85
6	17 828.65	35.04	59.58	35.66	1.29
7	4 280.48	2.18	24.59	8.56	3.61
8	4 366.36	2.50	23.68	8.73	3.53
9	1 165.05	0.71	4.55	2.33	0.95
10	14 768.56	26.52	69.49	29.54	4.63
11	1 995.39	2.25	18.38	3.99	1.37
12	8 681.85	7.06	55.34	17.36	4.82
13	1 243.99	1.57	3.80	2.49	0.34
14	54 239.08	40.44	951.48	108.48	77.91
15	16 956.75	32.57	52.64	33.91	1.40
16	11 599.28	22.61	37.22	23.20	0.95
17	26 634.03	45.56	2 787.87	53.27	122.43
18	6 629.72	12.66	19.13	13.26	0.37
19	2 185.70	4.25	16.51	4.37	0.55
20	2 242.31	4.37	4.84	4.48	0.08
21	2 187.51	2.18	15.48	4.38	1.32
22	3016 265.45	5 931.59	7 748.89	6 032.53	96.97
23	3145 763.88	6 001.16	9 759.22	6 291.53	470.36
24	22 305.63	17.80	440.86	44.61	26.09
25	33 961.62	20.13	1 746.28	67.92	149.05
26	5 250.83	1.83	41.22	10.50	5.62
27	4104 553.80	7 465.02	23 780.22	8 209.11	1 238.71

(a) Original statements set

Statement ID	Total Execution Time	Minimal Execution Time	Maximal Execution Time	Average Execution Time	Standard Deviation
12	5 222.33	4.34	38.27	10.44	6.28
17	18 382.34	35.20	56.35	36.76	1.04
24	19 443.67	15.01	335.98	38.89	23.95
25	27 682.09	17.93	1 365.19	55.36	123.57

(b) Index II statements set

Figure A.1: Partial results of Simple benchmark (time in ms)

Statement ID	Total Execution Time	Minimal Execution Time	Maximal Execution Time	Average Execution Time	Standard Deviation
2	4 212.73	7.87	25.36	8.43	0.97
5	3 183.05	6.28	8.86	6.37	0.16
6	3 453.29	6.65	14.16	6.91	0.41
7	4 260.58	2.39	20.21	8.52	3.47
8	7 417.70	14.53	27.58	14.84	0.61
10	4 539.04	7.25	11.95	9.08	0.60
14	46 671.10	27.06	467.25	93.34	68.82
15	17 376.07	33.46	55.15	34.75	1.12
22	2207 947.31	672.45	5 586.61	4 415.89	1 696.37
26	4 111.20	2.09	14.65	8.22	3.41
27	771.47	1.50	2.03	1.54	0.06
28	4 963.56	3.52	42.92	9.93	7.35

(a) Optimized statements set

Statement ID	Total Execution Time	Minimal Execution Time	Maximal Execution Time	Average Execution Time	Standard Deviation
1	1 307.06	2.54	27.06	2.61	1.09
2	2 725.99	5.25	7.79	5.45	0.19
3	1 179.96	0.98	30.83	2.36	1.42
4	1 769.47	1.91	4.86	3.54	0.13
5	1 830.38	3.61	4.04	3.66	0.04
6	1 999.57	3.79	8.50	4.00	0.27
7	1 305.25	2.54	3.02	2.61	0.05
8	1 381.33	2.72	3.12	2.76	0.04
9	545.96	0.92	2.59	1.09	0.44
10	3 177.93	4.33	18.64	6.36	0.85
11	1 268.81	2.51	3.18	2.54	0.04
12	6 086.03	4.60	399.99	12.17	20.49
13	1 198.11	2.08	3.21	2.40	0.26
14	44 291.08	23.66	895.07	88.58	71.54
16	1 508.14	2.97	10.92	3.02	0.36
17	19 061.94	36.87	60.76	38.12	1.49
18	5 572.01	10.52	16.59	11.14	0.44
19	1 126.23	2.17	4.24	2.25	0.09
20	434.90	0.71	45.49	0.87	2.20
21	1 359.70	2.33	5.50	2.72	0.17
23	13 180.01	14.23	87.93	26.36	9.42
24	19 409.28	14.37	408.24	38.82	26.75
25	29 118.42	17.68	2 383.23	58.24	163.51
26	1 072.04	1.95	2.60	2.14	0.07

(b) Index statements set

Figure A.2: Partial results of Simple benchmark (time in ms)

Appendix B

CD content

This appendix contains the list of files located on the enclosed CD and their brief description:

1. BenchmarkSimple.php – contains the code of Simple benchmark
2. BenchmarkComplex.php – contains the code of Complex benchmark
3. WithoutIndex.xml – contains Original statements set used in Simple benchmark
4. WithoutIndexOptimize.xml – contains Optimized statements set used in Simple benchmark
5. WithoutIndex.xml – contains Index statements set used in Simple benchmark
6. WithoutIndex.xml – contains IndexII statements set used in Simple benchmark
7. actions.php – contains Original actions set used in Complex benchmark
8. actionsOptimized.php – contains Optimized actions set used in Complex benchmark