Database usage in web page segmentation

Jan Zeleny

***Abstract*—Considering principles like templates, it is possible to reuse the web page segmentation results for more pages than just the one. Segmenting just one page and storing the result for other pages based on the same template can improve segmentation performance significantly, especially for methods based on visual appearance of the page. Storing the template along with the original page structure (and reusing it afterwards) can be thought of as a cache for segmentation algorithms. The motivation for this paper is to find out what structures and what features of these structures need to be stored for the cache to work as expected.**

***Index Terms*—object-oriented databases, object-relational databases, DOM tree, VIPS, vision-based segmentation**

I. Introduction

In recent years, the World Wide Web has become perhaps the most important source of information in the world. A family of algorithms for web-focused information retrieval grows with it. One step of information retrieval is understanding different parts of the web page. This is achieved by segmenting the web page and classifying resulting segments. Although the area of web page segmentation has been extensively researched, storing results of this process is usually discussed only marginally even though it can be used to achieve much better performance.

When we consider principles of modern web page design like templates, it is possible to reuse the segmentation result for more pages than just the one it was created with. The principle of templates defines that there is one template for a set of pages within the same site. This template contains blank spaces, where different data are placed to create different web pages. Segmenting just one page and storing the result for other pages based on the same template can improve segmentation performance significantly, especially for methods based on visual appearance of the page. Storing the template along with the original page structure (and reusing it afterwards) can be think of as a cache for segmentation algorithms. The motivation for this paper is to find out what structures and what features of these structures need to be stored for the cache to work as expected. This paper also describes different database types possibly convenient for storing desired structures.

In section II a brief introduction to the area of segmentation and classification is made, including basic comparison of different approaches. An analysis of the DOM model and different vision-based segmentation algorithms' output follows in sections III, IV-B and IV-B. The last thing that needs to be stored is described in IV-C. Sections V and VI describe use cases and the database for these use cases respectively. Description of some database types convenient for storing proposed database follow in VI-A and VI-B.

II. Web Page Segmentation

Web page segmentation algorithm are part of Information Retrieval field. Web pages are semi-structured documents. That means they have some structure but it is usually weak, so it is necessary to transform them to a form with hardened structure. The problem with this transformation is that web pages usually contain more logical parts and most of these parts don't contain any useful information. For example news servers contain navigation, polls, advertisements and links related articles. If we wanted to index such page for example for purposes of web search engine, it is important that only the useful data is used and misleading information is filtered. Although this is the main goal of majority of segmentation algorithms, other applications are possible as well – for example finding the navigation in order to construct site map or filtering advertisements.

The goal of visual segmentation algorithms is to identify logical blocks as user sees them on the page. To achieve this, the page has to be rendered first. That is usually very demanding operation in terms of computational time. However these algorithms have great potential. The pioneering work

in this area has been done by Cai et al. Their algorithm VIPS [2] is used in many subsequent papers dealing with page segmentation or classification of identified segments. Only a few algorithms not using VIPS are being developed. One of them was introduced by Burget et al. [10]. Although they work differently, some concepts remain the same. In further text we will be focusing on the latter algorithm and its result, but results of the former one will be briefly described as well.

As it was already outlined, there are couple problems with vision-based segmentation algorithms. The main one is their speed. The other one is precision. In certain areas they perform very well, but their results are not always satisfying. The latter one might be solved with more complex algorithms, but this would mean even more computational time needed for the segmentation. The time issue can be solved by using different algorithm family. There are two main alternatives to visual segmentation. The closest one is the DOM based segmentation. Algorithms in this family only analyze features of the DOM tree and its nodes like content and tree-related features. Also some basic visual features are analyzed, but not in the scope as that of vision-based algorithms. Examples of different DOM-based algorithms can be found in [17] and [16].

Another alternative approach is the Template Detection [18], [22], [20], [24], [23], [19], [21]. Template is one of fundamental principles used on modern web sites. These are not made of a large amount of distinct pages, but rather from a set of templates. This concept brings easier web page creation and organization, therefore it is mainly used in large CMSs, e-shops and blog applications. Overall it is possible to say that the larger the site is, the greater is the likelihood of templates being utilized. Of course larger site usually means more templates are used. That's why these methods often contain a clustering step – each cluster represent a group of pages using the same template. Even with this step, these methods usually scale better than algorithms described above – instead of segmenting one page at a time, more pages are analyzed in parallel and the result is then applied to all of them. What is important about template detection is that these algorithms usually segment pages only in two parts: *template* and *content*. Template contains all information besides the content and it is not segmented further. That's why these algorithms can't be used as 100% replacement for vision-based segmentation algorithms.

The goal of this work is to utilize advantage template detection algorithms give us against visual segmentation while removing their weaknesses like the requirement to have certain amount of web pages for the algorithm to work and segmenting only in two parts.

III. DOM TREE

The Document Object Model (DOM) is both language and architecture independent model used to represent SGML-based documents. It is especially used in area of World Wide Web as a means to work with XML, XHTML and HTML documents. It is the most common model used for this purpose. The model has been standardized by organization W3C in 1998 and it has developed and extended considerably since then. It defines several interfaces from the standard core, HTML and XML to CSS and event handling. The standard has been developed in three phases (which however did not follow exactly one after another) called *Levels*:[1], [6]

- **Level 0** only an informal specification containing a mix of HTML document functionalities. It is sometimes referenced in further specifications for backwards compatibility.[7]
- **Level 1** contains the core DOM specification. It also contains some parts covering HTML and XML, but the main focus is on the Core.
- **Level 2** brings definition of handling style sheets and style information attached to particular nodes. Some parts of Core definition from Level 1 were updated as well. It also brings other features, but they are not important for this article.
- **Level 3** does not bring anything important for this article. Only some updates of Core module are presented.

The following text will focus only on interfaces defined in Core, HTML/XML and CSS parts.[1]

The tree is described by its API, which has been implemented in many languages including Java, JavaScript, PHP and C#. It can be used to describe all content, structure and visual style of the web page. The API itself consists of more interfaces, each element can be described by one or more

[1]Note that many principles described also apply to interfaces outside our scope.

of them. All interfaces can define both properties and methods. Since methods are not stored in the database, only properties will be important to us. The specification mentions two approaches to the interface representation and implementation. The first option is to follow classical object-oriented concept of inheritance. The second one is a concept of flattened view of the API. That means all properties and methods of each object can be accessed by its base class without the need of casting.

There are four basic data types in the specification:

- **DOMString** a sequence of two-byte units
- **DOMTimeStamp** this is used to store absolute of relative time in a form of integer number measuring in milliseconds
- **DOMUserData** represents a reference to application data. It can contain virtually any data necessary.
- **DOMObject** is a reference to any object (equivalent of Object type in Java)

Besides these there are also some extended data types like collections and lists (ordered collections). DOMStringList and HTMLCollection are typical representatives of these. Items in both can be accessed by ordinal number starting from zero, therefore both are basically equivalent to array type known from many languages.

Now let's summarize some general properties of the DOM tree with regard to storing it in the database.

- each node can have N child nodes depending on its type.
- a node can have multiple attributes, again based on its type. These attributes can be represented either by one of basic data types or by child nodes of Attr type. In this approach all attribute objects are not considered to be a part of the DOM tree. HTML extension of the DOM also specifies that attributes can be present as properties of the element object. However this approach is deprecated and is specified only for compatibility reasons.
- sibling nodes are linked in the list. They are accessible as described above for collections but in addition they are double-linked themselves (each node has a link to previous and next sibling node).
- text is always considered to be node as well (otherwise it would be impossible to represent for example text with bold parts)

Here follows a description of some basic object classes (interfaces) defined by the Core module including their brief description why they could be stored:

- **Document** describes the document and its properties like its URI, document type and encoding. It also contains reference to the root node of the DOM tree.
- **Element** represents a tag in the document
- **Attr** in case attributes are represented by this object. It contains only three useful informations: name and value of the attribute and a flag if the attribute was specified or not.

Since this paper discusses storing the DOM tree mainly for the purpose of mapping it to the output of segmentation algorithms, it is possible not to store the content of DOM tree itself. That means the text nodes don't have to contain the text itself. Also image nodes don't have to contain the image data. But because some heuristics can be performed on the stored tree, it is reasonable to save at least some properties representing the text or image. These can be for example character/word count or image dimensions respectively.

As for CSS support, DOM offers two interface families. The first one is designed to attach style sheets to documents. That's not important for our purpose. However the second one focuses specifically on CSS related properties of document and particular elements. The most important part is defined in CSS2 extended interface, which basically states that each CSS attribute has its own equivalent in DOM. For example `margin-right` CSS attribute has `marginRight` as its equivalent. All these DOM equivalents are grouped in DOM attribute `style`, which each DOM node representing HTML element can have.

IV. Tree of visual areas

For the purpose of this paper let's consider the output of every visual segmentation algorithm to be the Tree of Visual Areas. Different algorithms have different output formats, but they all have characteristics of the tree structure. Considering this and further focus of this paper these output structures will be described in a way which shall bring them close to formats needed for databases

in further sections. Now let's see output formats of different algorithms.

A. VIPS

VIPS stands for Vision Based Page Segmentation[2]. After being processed by VIPS[2], the web page is represented by a set of blocks, a set of separators and a relation between blocks.

The most important feature of blocks is that they are not overlapping. This means when put together they complete level of the resulting tree (i.e. their union creates their parent block). Each block in the set is recursively segmented and then represented by another set of blocks, separators and relation. This implies a tree-like structure of the whole construct. It also means that the web page itself is considered and treated the same as any other visual block. Leaf nodes of the resulting tree are called *basic objects*. Each basic object corresponds to one node in the DOM tree. Therefore each visual block can contain one or more nodes of the DOM tree. Note that the Tree of Visual Areas and the DOM tree don't have to correspond, i.e. a visual block doesn't have to correspond to a particular node in the DOM tree. Because some additional heuristics and classification algorithms are performed on segmentation results in some algorithms[9], [8] and it is also necessary for the cache to work, the corresponding DOM tree and the mapping between both trees should be stored along with the output of VIPS.

For each block an information about its position and size is absolutely essential. These properties can be expressed as absolute numbers or relative to the parent block[9]. Also the alignment with its parent (for example float left) is used[9]. Considering how VIPS works, an information about the *Degree of Coherence* as defined in [2] should be stored for each block as well. For separators is is important to store their visual impact, which can be in form of width or visibility defined e.g. by borders or background color of adjacent blocks. Relations between blocks have one feature and that is the degree of visual similarity of blocks in relation. This information is not a part of VIPS output, but it is added in some other algorithms using it[9]. The last piece needing description is the relation between blocks on one level of the tree. Two blocks are in this relation if and only if they are adjacent to each other. An important part of this relation corresponding also a little with visual impact of a separator is the degree of similarity of each 2-tuple in this relation[9].

B. Other algorithms

VIPS is not the only algorithm producing a Tree of Visual Areas as its output. Burget in his work [10] focuses on similar problems as VIPS. Structures described in his work are slightly different. They are also described in some of his previous works which are not referenced here (they are not in English).

The tree produced by his algorithms contains two node types: *visual areas* and *content nodes*. All nodes except for the leaves are visual areas. Leaf nodes of the tree are content nodes, also known as content blocks. For the next paragraph we consider the tree without these content nodes. All visual areas contain information about the position and dimensions of the area. To define both of these a special topographical grid is constructed for each non-leaf visual area. En example of this grid is displayed on figure 1

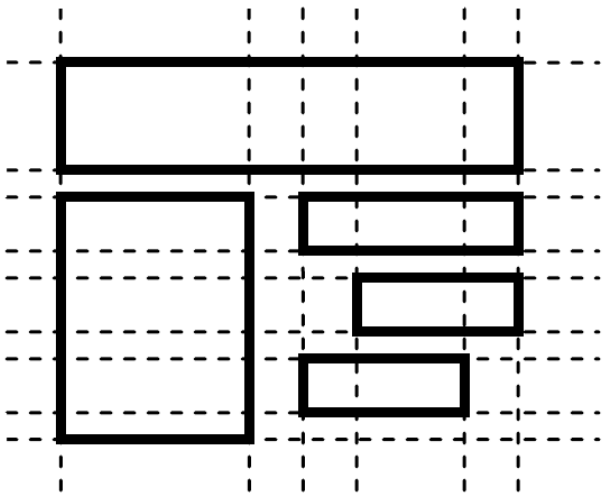

Fig. 1. An example of the topographical grid

All child areas are then placed in the grid. A position of each area is represented by the cell of grid which the top-left corner of the area is in and dimensions are represented by the number of rows and columns the area takes. Here it is important to remind that each area is rectangular by definition. Every non-leaf node in the tree can contain only other visual areas as child nodes. Each leaf contains exactly one content node, therefore no grid is necessary for it.

Content nodes, also known as content boxes, contain one or more content elements. Each content node represents visual element on the page. In case it contains more content elements, they are concatenated to a string creating a single continuous area of the document. Content element is an abstract denotation. A way how to create content boxes when composing the web page is specified by W3C standard. There are two types of content element:

- **image element:** contains only information about image width and height and the actual image data in an arbitrary image format
- **text element:** contains the actual text and properties of that text such as font size, color, style, weight, ...

A root of the Tree of Visual Areas is created by a Document node. This is not listed as the third node type, because it is in fact a visual area with a title.

C. Mapping of trees

As discussed in previous sections, it is important to save both the original DOM tree and the Tree of Visual Areas. It is also important to know which nodes of the DOM tree are represented by particular visual area.

An example of typical tree mapping is displayed on figure 2.

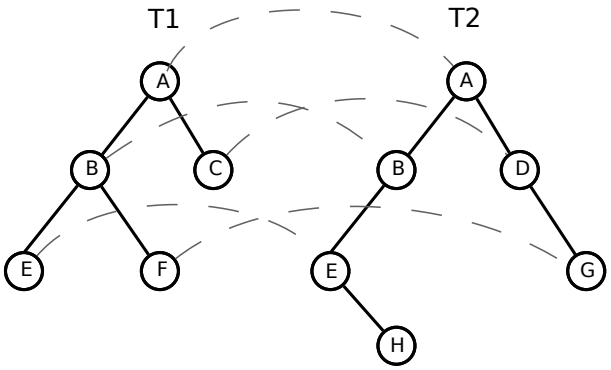

Fig. 2. An example of tree mapping

Mapping of tree for our purposes is different than the problem of tree mapping as described in [11] and [12]. In the literature, it is intuitively defined as a list of actions (add, delete, replace) needed to transform one tree into another, but in our case we just need to know which nodes of the DOM tree are represented by a particular visual area in the Tree of Visual Areas.

Each leaf node in VIPS output is mapped 1:1 to a node in the DOM tree. This can be solved by simple object reference. However non-leaf nodes don't have to correspond to particular nodes. To find out which DOM nodes the area represents, a recursive search for all leaf successors and their corresponding DOM nodes can be performed. However for convenience it should be possible to store a list of subtrees represented by the visual area. A two-way reference between DOM nodes and visual areas might be considered. That would have to be M:N if we consider that more Trees of Visual Areas might be derived from single DOM tree.

Mapping between DOM tree and Burget's algorithm output tree is similarly simple. Content nodes are represented by their DOM counterparts as described in sections IV-B and III. The rest of the mapping is fairly the same.

V. Segmenting chain

Now that all structures which will need to be stored are analyzed, it is important to specify some things about the whole segmenting chain in order to have complete information for the database design.

A. Parsing sources

In the first step, HTML and CSS sources are parsed and the DOM tree is created. The tree can be represented for example by standard classes in Java, available in `org.w3c.dom` package.

B. Cache checking

This step has to be done right after the DOM tree is created in order to spare as much computational time as possible. For the cache checking a simple algorithm can be used, for example the path comparison as described in [22]. For that purpose a set of DOM paths in the document needs to be stored or at least simply calculable.

C. Cache hit

In case the document template has been found in the cache, corresponding tree of visual areas has to be easily retrievable as same as the mapping between the tree and the template. So it is clear which visual area contains specific DOM node and vice versa.

D. Cache miss

In case the document isn't found in the cache, it has to be segmented and the result along with the original DOM tree and the mapping has to be stored.

VI. Database

Based on analysis in previous sections, it is now important to define structures that will be stored in the database. The DOM element is described as follows:

```
public class DOMNode
{
  String name;

  LinkedList<DOMNode> children;
  Map<String, String> attributes;
  HashSet<VisualBox> boxes;

  String path;
  DOMNode nextLeaf;
}
```

Name means the name of the HTML tag which the node represents. The list of children in obvious, just a note here that the list has to be linked in order to achieve given order of sibling nodes which won't change. Attributes are here just to complement the design, they are not needed for anything in particular. They can be however used for more precise comparison of DOM trees if necessary. The Last two attributes are here only to outline how it is possible to achieve a simple computation of path set of particular DOM tree. Each leaf will store complete path leading to it and the link to the next leaf node. The set of paths will be then constructed by simple iteration over single-linked list.

The DOM document is described as follows:

```
public class DOMDocument
{
  String url;
  DOMNode root;
  VisualArea segmentation;

  HashSet<String> pathSet;

  DOMNode firstLeaf;
}
```

URL can be used for simple duplicate detection. It can be also used for template matching as described in The `root` attribute basically points to the tree itself. The last two attributes outline two approaches which might be considered for path set storing/computing. The first consists of computing the path set directly while assembling the DOM tree and storing it along with the tree. The second one was already described above. This is just a marker for the first leaf in the tree (pre-order traversal).

Visual areas and visual boxes will be stored in following structures. The design is based on original XML output of Burget's algorithm. One important thing is how visual boxes, visual areas and DOM nodes are linked to each other. If badly designed, a problem could occur with circular dependencies. Some databases don't handle these situations well, therefore circular dependencies have to be reduced to minimum. In following design all circular dependencies are eliminated, because there is only forward linking in a form of `elements` attribute of `VisualArea` class. The linking is represented this way in order to optimize probably most used operation – retrieval of all DOM elements contained in visual area of specific type.

```
public class VisualArea
{
  String id;

  int x1; int y1;
  int x2; int y2;
  int gx1; int gy1;
  int gx2; int gy2;
  int gridw; int gridh;

  String background;
  float fontsize;
  float fontweight;
  float fontstyle;

  LinkedList<VisualArea> children;
  LinkedList<DOMNode> elements;
}
```

```
public class VisualBox
{
  int x1; int y1;
  int x2; int y2;

  String color; String fontFamily;
  int fontSize; String fontVariant;
  int fontStyle; int fontWeight;
  HashSet<String> decoration;

  boolean replaced;
  VisualArea parent;
}
```

Now that structures to be stored are defined, the next step is to analyze databases where these structures can be easily stored.

A. Object-relational databases

The Object-Relational data model is partially derived from common relational data model, but in comparison it solves their biggest weakness and that is the atomicity of attributes as defined in the *first normal form*. This limitation is not a problem in simple applications (in a terms of precessed data) such as banking systems. However even there it is possible to observe first complications. For example addresses are usually represented by several fields (street, house number, city, etc.), but the application might have a need for the address as one corpus. O-R data model solves this by introducing the possibility to store *user data types*. Address would be just stored on one field of the table and it would be represented by an object holding all these information separately. Following block of code shows an example how could the data type be defined:

```
create type address as
(
  number integer,
  street varchar(100),
  city varchar(100),
  zip integer
)
```

Each property of object of user data type can be reached by a "dot notation" as shown in the following example

```
select addr.city from users
```

After being defined, objects of the data type can be used in data fields of tables and as properties of other objects. To get closer to the object-oriented paradigm it is also possible to define tables of the type as demonstrated by following example. Objects of defined type will then represent rows of the table. This concept is also required for some features which will be described later.

```
create table addresses of address
```

User defined types in SQL are equivalent to classes in standard programming languages. As such, they have some features known from these languages. One of such features is *type inheritance*. When inheriting a type, both methods and properties are inherited. Inheriting methods has its specifics, but, since methods are not important for our purposes, they will not be discussed here any further. It is possible to use keyword `final` to specify that a type can't be inherited any more. SQL defines not only type inheritance, but also *table inheritance*. This corresponds to generalization/specialization as known from entity-relationship models. Both these inheritances imply the possibility to use the hierarchical DOM model design as described in section III.

Reference types are the next useful feature of object-relational databases. They are very similar to object pointers in C++. It is possible to store a reference to another object as an attribute of an object. To use this in SQL, there is one limitation: there has to be a table of referenced object type in the database and of course the referenced object has to be stored in this table. When creating a type which contains reference attribute, this attribute has to be given its scope – the table containing objects it is possible to reference in that attribute. The scope is mandatory and it makes this whole concept work similar as foreign keys in relational databases. The table of referenced type has to have *self-referential attribute* defined – this is similar concept as in standard foreign keys. Dereferencing of an object is similar as dereferencing pointers in C++. We can use either `->` operator, or combination of `deref()` and `.` operators. Reference types are important for modeling the mapping between trees, because each node can be referenced both from the tree itself and also from the tree of visual areas.

Besides user defined data types, object-relational databases offer two new data types: *arrays* and

multisets. Multiset is a type similar to *set* known from standard SQL, but it can contain multiple same values as well. It was first defined in SQL 2003. The array type is basically the same as known from C language – it also has to have pre-defined length and can store only one data type. As of SQL 2003, the array can be also defined as unbounded. In that case the only boundary is defined by implementation. Compared to relational databases, both arrays and multisets simplify some common design problems. Simple 1:N relations are a good example of this.

There is one more simplification, which can be considered new data type: *unnamed row*. These are an alternative solution to user defined data types. For example when we consider the `users` table described above, unnamed row can easily replace the `address` data type in `addr` column as example below shows. This might be useful for storing attributes of DOM nodes assuming that the flat DOM design is used.

```
create table users
(
  addr row
  (
    number integer, ...
  )
)
```

The best approach to store DOM attributes is in an associative array or as hash type, used for example in Java. This type can be emulated to some degree by a database table, but the result would not be entirely satisfying. Alternative approaches for this can be either array of objects representing DOM attributes or specific column in the table for storing DOM elements for each attribute. The latter one and the associative array emulation are also possible options to store style information as defined in the DOM specification.

Since object-relational databases support object references, it is also possible to create more complex data structures like lists, queues and stacks. Specifically, the first one can be used to store relations between VIPS blocks by means of objects containing references to both VIPS blocks they separate. The positional grid as described in [10] can be designed similarly, as two-dimensional array of grid cells. This array will be also implemented with object references. There is one more issue of object-relational data model left and that is bridging the gap between programming language of an application and the language of the database. This issue is closely described and dealt with in section VI-B.

B. Object oriented databases

The concept of object oriented databases as described by Silberschatz et al. in [14] was designed to solve the biggest issue of relational and object-relational databases and that is transformation of the data from typeset on the database to the typeset of the programming language the application itself is written in. Language used in these databases (SQL or similar) is usually not convenient for writing the entire application. Particularly, the user interface is almost always written in another language. The process of converting the data has two flaws. First of all, it takes a substantial amount of code. That means less lucidity and greater likelihood of an error in the code. Also the persistence has to be handled explicitly – when the data is modified in the program, a routine to store them in the database has to be called and its result evaluated. Again, it means a lot of additional code.

Object-oriented databases are based on a concept of *persistent programming languages*[14]. In these languages the query language and other means of data handling are integrated into the application language, therefore the typeset of database and application is the same – no additional conversion is needed. Persistent programming languages are basically standard programming languages like Java with framework handling the persistence of some objects. This means that *there are no issues or missing features of such databases*, because they share feature set with language of the application and therefore all constructs and object previously proposed can be created and used. There are, however, some features related to data storing which should be evaluated.

Normally, objects are transient and they disappear once the program is terminated. There are four approaches how to make transient objects persistent[14]:

- *persistence by class*: in this approach the whole class is defined as persistent and all objects of this class will be automatically stored on the disk. This approach is not convenient for our purposes, because we might need some

temporary objects upon which many operations need to be performed before they are ready to be stored.
- *persistence by creation*: here the object is marked to be persistent when it is created. This approach is slightly better, but still inconvenient for use for the same reason as the previous one.
- *persistence by marking*: this is the first approach which might be used. Objects are created as transient and they are marked as persistent at any point of their life
- *persistence by reachability* is probably the best option for storing DOM tree and the Tree of Visual Areas. Here a root object is marked as persistent and all objects become persistent once they are reachable from this root object. Also breaking their reachability from the root makes them transient again. In this approach it is possible to create nodes of the tree as fit and after all preparations are performed on them, they can be connected to the tree as needed and thus made persistent. Also in this approach it is easy to dispose larger subtree by simply disconnecting a single node – the root of that subtree – from the tree.

What is important in object-oriented databases is the object identity and its persistence. Transient objects have their identity very straightforward. The identity corresponds with object's position in the memory (for example in a form of pointer to the object). However position of persistent objects may change in time. We need to know how to refer to these objects when for example the program ends and starts again later. There are four levels of identity persistence[14]:

- *within the procedure:* basically no persistence at all, local variables can be an example of this. For our purposes it is not usable.
- *within the program:* this level of persistence can be used in some specific cases described below. In other cases it corresponds to global variables for instance.
- *between programs:* this corresponds to pointers to file system. The problem here is that these pointers can be changed in time. That can cause the stored tree to fall apart.
- *persistent:* in this case the identity survives even data reorganization on the disk. It is, therefore, the optimal level we can achieve.

There are also some drawbacks to using persistent programming language[14]. The biggest one is that programming languages used for this are usually high-level. That means worse optimization of performed operations and rather big overhead. Historically, there was also worse support of declarative querying, but lately a significant progress has been made in this area.

There are many implementations of object-oriented database concept. Because Burget's web segmenting application is written in Java and the proposed data structures were also designed in Java, we'll focus on interfaces supported there. The most known interfaces for persisting Java objects are JPA and JDO. However, neither of them is object-oriented database per se, they are just interfaces for storing Java objects in various database storages. In addition, there is usually a problem with performance, since the typeset conversion is done within engines implementing these interfaces. There are many implementations of pure Java object storage, however with some constraining conditions like client-server architecture, only a handful of them remains.

One of them is NoSQL database OrientDB. It is foremost document oriented database, but object-oriented wrappers are provided as a part of the project. Basically these wrappers very simply translate plain Java objects to documents, which are then stored in the database. Java reflection is used to find information about Java objects and therefore it doesn't need almost any code enhancements or proxy classes as used by JDO and JPA implementations.

Now just to list some features related to previously described features of object-oriented databases. OrientDB typically persists objects by marking them as persistent. This is done via `save()` method of *database descriptor*, which handles persistence. Following example [15] demonstrates both persistence by marking and persistence by reachability (the newly created `City` object).

```
String dbpath =
  "remote:localhost/petshop";
ODatabaseObjectTx tx =
  new ODatabaseObjectTx(dbpath);
ODatabaseObject db =
```

```
  tx.open("admin", "admin");

Person p = new Person();
p.setName("Luca");
p.setSurname("Garulli");

db.save(p);
db.close();
```

This is perfect for our purpose, since it is possible to build the entire tree and then mark its root as persistent. This will store the entire tree. As for object identity, OrientDB objects have identity persisted on the disk along with the object data. That means it is entirely persistent. Only a small modification in the code defining classes is necessary, Each persisted class has to contain two `Object` type properties annotated as `@Id` and `@Version`, which will contain the object identity. The latter one is needed only when working with transactions to allow multi version concurrency control. As for performance options, OrientDB offers both client-server and embedded approach. It also offers many option for performance tuning like cache size transaction tweaks, and index finetuning. It also supports custom indexes defined on object attributes. Lazy retrieval is supported as well as eager one. Based on some experiments performed on proposed object structures in the database, the only big issue are circular dependencies between objects – they are handled very poorly by the DB and thus they can slow down persistence of a DOM tree by thousands of percent. The proposed design is already modified solution which doesn't suffer with this.

VII. Conclusion

In the first half of this work, outputs of different vision-based segmentation algorithms have been inspected and described in a way which brings them closer to object-oriented tree-based approach. Also a DOM model has been described with special attention to those of its features which are important to store. The last data component inspected was the mapping between the two trees. During the inspection of all data structures, some aspect of their design were emphasized as potentially problematic for storage.

In the second part, database structures were proposed. Some of them were already tested and the result represents optimized data structure set in terms of both stored information and performance the cache considering the main use cases. These use cases were described prior to proposing database structure.

Then basic features of two database approaches convenient for storing the data identified in the first part were outlined. Each one of proposed database approaches has its specifics. In comparison Object oriented databases offer much better design possibilities, making most of the structural specifics possible to design and implement. From this point of view they are more convenient for data storage. Of course not every language offers the persistence extension, therefore object-oriented approach is not always the best option. Also some performance issues are likely to occur in comparison with object-relational data model. If necessary, this model can also offer some features making the database design needed for described structures possible. But the final design would have several shortcomings which would need to be compensated by additional program code. Therefore it won't be the first choice in most cases. Described OO DB engine, the OrientDB, offers all features which are needed for this design. It also presents very simple interface and, based on some preliminary tests, it shows promising result in terms of performance and scaling.

References

[1] Cover, R.: Technology Reports: W3C Document Object Model (DOM). 2003

[2] Cai, D., Yu, S., Wen, J.-R., Ma, W.-Y.: VIPS: a Vision-based Page Segmentation Algorithm.. Microsoft technical report. MSR-TR-2003-79. 2003

[3] Hors, A. L., Hgaret, P. L., Nicol, G., Wood, L., Champion, M., Byrne, S.: Document Object Model (DOM) Level 3 Document Object Model Core. W3C Recommendation. April 2004

[4] Stenback, J., Hgaret, P. L., Hors, A. L.: Document Object Model (DOM) Level 2 Document Object Model HTML. W3C Recommendation. January 2003

[5] Wilson, C., Hgaret, P. L., Apparao, V.: Document Object Model (DOM) Level 2 Style Specification. W3C Recommendation. November 2000

[6] W3C: Document Object Model (DOM) Technical Reports (overview)

[7] Hors, A. L., Wood, L., Sutor, R. S.: DOM specification, Glossary. January 2003

[8] Petasis, G., Fragkou, P., Theodorakos, A., Karkaletsis, V., Spyropoulos, C. D.: Segmenting HTML pages using visual and semantic information. In Proceedings of the 4th Web as a Corpus Workshop, 6th Language Resources and Evaluation Conference. June 2008.

[9] Liu, W., Meng, X., Meng, W., ViDE: A Vision-Based Approach for Deep Web Data Extraction. IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 3. March 2010.

[10] Burget, R.: Layout Based Information Extraction from HTML Documents. The Ninth International Conference on Document Analysis and Recognition. 2007.
[11] Tai, K. C.: The tree-to-tree correction problem. J. ACM 26(3). 1979
[12] Vieira, K., Carvalho, A. L. C., Berlt, K., Moura, E.S., Silva, A. S., Freire, J.: On Finding Templates on Web Collections. In Journal on World Wide Web, Volume 12 Issue 2. June 2009.
[13] EJB 3.0 Expert Group: JSR 220: Enterprise JavaBeans™, Version 3.0; Java Persistence API. May 2006
[14] Silberschatz, A., Korth, H. F., Sudarshan, S.: Database System Concepts, 5th edition. New York: McGraw-Hill.
[15] Garulli L. et al: http://code.google.com/p/orient/wiki/ObjectDatabase. Dec 2010
[16] Lin, S. H., Ho, J. M.: Discovering Informative Content Blocks from Web Documents. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2002
[17] Laber, E., Souza, C., Jabour, I., Amorim, E., Cardoso, E.: A Fast and Simple Method for Extracting Relevant Content from News Webpages. In *Proceeding of the 18th ACM International Conference on Information and Knowledge Management*. 2009
[18] Yi, L., Liu, B., Li, X.: Eliminating noisy information in Web Pages for Data Mining. In *Proceedings of the International ACM Conference of Knowledge Discovery and Data Mining*
[19] Reis, D. C., Golgher, P. B., Silva, A. S., Laender, H. F.: Automatic Web News Extraction Using Tree Edit Distance. In *Proceedings of the 13th International World Wide Web Conference*. 2004
[20] Valiente, G.: An Efficient Bottom-Up Distance between trees. In *Proceedings of the International Symposium on String Processing and Information Retrieval*. 2001
[21] Vieira, K., Silva, A. S., Pinto, N., Moura, E. S., Cavalcanti, J. M. B., Freire J.: A Fast and Robust Method for Web Page Template Detection and Removal. In *Proceedings of the Acm International Conference on Information and Knowledge Management*. 2006
[22] Gottron, T.: Bridging the Gap: From Multi Document Template Detection to Single Document Content Extraction. In *Proceedings of the IASTED Conference on Internet and Multimedia Systems and Applications*. 2008
[23] Vieira, K., Carvalho, A. L. C., Berlt, K., Moura, E. S., Silva, A. S., Freire J.: On Finding Templates on Web Collections. Springer Science + Business Media. 2009
[24] Yossef, Z.-B., Rajagopalan, S.: Template Detection via Data Mining and its Applications. In *Proceedings of the International Conference on the World Wide Web*. 2002