Storing results of web page segmentation
(Evaluation of post-relational databases)

Jan Zelený

FIT, Brno University of Technology, Czech Rep., email: `izeleny@fit.vutbr.cz`

Abstract. Segmenting the web page is one of initial steps of information retrieval process performed on that page. While there has been an extensive research in this area, storing the results is usually only a marginal topic. 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, thus improving the performance of web site processing engines. This paper analyzes what structures have to be stored and what databases are convenient for this purpose in order to enable and support reusing the segmentation algorithms' results.

1 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 for achieving 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. The motivation for this paper is to find out what structures and what features of these structures need to be stored, so they can be reused for further site processing as easily as possible. This paper also focuses on different database types convenient for storing desired structures.

Sections 2, 3 and 4 analyze what data do we need to store when segmenting page with algorithm based on visual appearance of the page. Section 5 then analyzes databases capable of conveniently storing tree structures and summarizes those of their features which might be convenient and/or inconvenient for storing structures identified in previous sections.

2 DOM tree

The Document Object Model (DOM) is both language and architecture independent model used to represent SGML-based documents. The tree representing a web page is described by its API which can be used to describe all content, structure, and visual style of the web page. The API itself consists of many interfaces, each element can implement one or more of them. All interfaces can define both properties and methods, but only properties are to be stored. There are two allowed approaches to the interface representation and implementation. The first one is to follow common object-oriented concept of inheritance. The second one is a concept of flattened view of the API (no interface hierarchy).

There are four basic data types in the DOM tree: (1) string, (2) timestamp represented by integer number, (3) user data blob and (4) object which is a reference to any other DOM object. For storing the tree, strings, integers and object references, these types should be sufficient. There are also some extended data types like collections[3] and lists (ordered collections)[2]. Items in both can be accessed by ordinal number starting from zero, therefore both are basically equivalent of common array type.

Following are some general features of the DOM tree which should be considered when storing it. First of all each node can have N child nodes (depending on its type). Similarly, a node can have multiple attributes (again based on its type). These attributes can be represented either by one of basic data types as element properties (deprecated) or by child nodes of Attr type. Sibling nodes are linked in the list. They are accessible like standard list items, but in addition each node has a direct link to previous and next sibling. When using hierarchical API model, a node can implement properties from multiple interfaces. A text is always considered to be node (otherwise it would be impossible to represent for example text with bold parts). 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.

Although it contains the means necessary to describe HTML pages, the Core module itself is not entirely convenient. HTML extension was designed as a layer providing this convenience by defining types of common elements and their attributes – it is no longer necessary to store them as objects of Attr class.

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.

3 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 similar characteristics, for example the tree structure.

3.1 VIPS

After being processed by VIPS[1], the web page is represented by a set of blocks, a set of separators and a relation between blocks (two blocks are in relation if they are adjacent).

The most important feature of blocks is that they are not overlapping. Each block in the set is recursively segmented and then represented by another set of blocks, separators and relation. This implies the tree structure of the whole construct. It also means that the web page 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 it is used in some algorithms[6, 5], 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[6]. Also the alignment with its parent (for example float left) is used[6]. Considering how VIPS works, an information about the *Degree of Coherence* as defined in [1] 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 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[6].

3.2 Other algorithms

Burget in his work [7] focuses on similar problems as VIPS but structures he uses are slightly different. The tree produced by his algorithms contains two node types: *visual areas* and *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. An example of this grid is displayed on figure 1

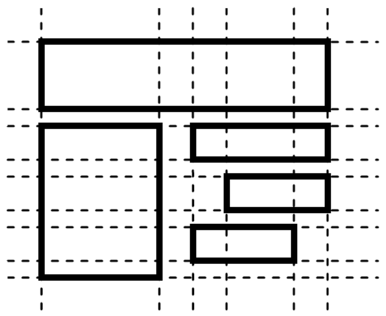

Fig. 1. An example of the topographical grid

All child areas are then placed on 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. Every non-leaf node in the tree can contain only other visual areas. Each leaf contains exactly one content node, therefore no grid is necessary for it.

Content nodes contain one or more content elements concatenated to a string creating a single continuous area of the document. There are two types of content element: *images* and *text*. Each of them contains different attributes describing appearance. Taking description of text nodes in section 2 into account, it is possible to consider text and image nodes in the DOM tree as content elements, because they are basically equal.

4 Mapping of trees

Mapping of tree for our purposes is different than the problem of tree mapping as described in [8] and [9]. In the literature, it is defined as a list of actions needed to transform one tree to another, but 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 3.2 and 2. The rest of the mapping is fairly the same.

5 Databases

Some design challenges were introduced in previous sections. This section will analyze different database types suited for storing previously introduced tree structures. In each part a special attention will be given to features which might be difficult to conveniently implement in that particular database. Only object-oriented and object-relational databases will be inspected here, because they support a basic feature needed for storing tree structures – objects and references to objects. Also a declarative access to data, which both database types offer, is strongly preferred.

A very brief summary of specifics we have to evaluate follows:

- Flat or hierarchical DOM node representation (hierarchical is preferred)?
- Is it possible to store references to N children? How?
- How can we store attributes (object properties or Attr data type)?
- How can we store lists (sibling nodes, DOM attributes, . . .)?
- How can we represent style attributes?
- How to store the relation between VIPS blocks?
- How to represent the grid in Burget's tree?

5.1 Object-relational databases

Object-relational data model is the first one offering the required set of features. It 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 )
```

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 2.

Reference types are the next useful feature of object-relational databases. They are equal for example to pointers to objects 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. Value of this attribute will then serve as the reference itself – it will identify the referenced object within the table. It is the same concept as primary key in relational databases. In fact existing primary key of the type can be used as self-referential attribute. 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 in the mapping.

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 same values multiple times as well. 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. Compared to relational databases, both arrays and multisets simplify some common design problems. Simple 1:N relations are a good example of this. However for our purposes, the pre-defined array size makes it unusable, because each DOM element can have arbitrary number of children as well as attributes. The only option how to design this is to use double linked list. Each node would have to reference only the first child. This approach, however, cannot be applied to mapping as described in section 4. The only option would be to reduce the mapping to 1:N, which is also described in section 4. The array limitation also affects attribute representation. Because hierarchical DOM model can be supported, it is reasonable to store the most common attributes as properties of data types. The rest of attributes can be represented for example by a list of Attr objects. The best approach would be to store attributes in an associative array which can be emulated to some degree by a database table, but the result would not be entirely satisfying. This also implies that the best way to store style information is as the DOM specification suggests – each style attribute will have its property in `style` data type.

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, ... ) )
```

There are the last two things left for evaluation: considering properties offered by object-relational databases, the best way to store relation between VIPS blocks is to store it as a list of `relation` objects. Each such object would have to have two `next` pointers and two `prev` pointers, one for each block in the relation. The grid design will be more of a challenge taking our options into account. The best option to design it would probably be as an array of objects linked similarly as a list. 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 5.2.

5.2 Object oriented databases

The concept of object oriented databases was designed to solve the biggest issue of relational and object-relational databases and that is transformation of the data from typeset of the database to the typeset of the programming language the application itself is written in. The process of data conversion has two flaws. First, it takes a substantial amount of code. That means less lucidity and greater likelihood of an error in the code. And second, 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*[11]. 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. This means that *there are no issues with features of such databases*, because they share feature set with language of the application and therefore all things we need to evaluate are supported. 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[11]:

- *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.

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. 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[11]:

- *within the procedure:* basically equals to no persistence at all
- *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[11]. 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 many web segmenting applications are written in Java, we'll focus on one specific interface: *Java Persistence API*[10] and its most spread implementation – Hibernate. It uses object-relational mapping concept. That means it is not database engine per se. Instead, it uses arbitrary SQL-based database as backend and then transforms the data from the database to Java objects. There is a set of well defined rules how to declare classes to correspond with the database layout (or vice versa). Several code/database generators are even based on this set of rules. Since the Hibernate framework does all the transforming itself, two main issues of object-relational and relational databases mentioned at the beginning of this section disappear.

Now just to list some features related to previously described features of object-oriented databases. Hibernate typically persists objects by marking them as persistent. This is dona via `persist()` method of *entity manager* object, which handles persistence. To some degree, persistence by reachability is also supported. In following example, object `a` is persisted by marking and object `b` is persisted by reachability:

```
A a = new A();
B b = new B();
a.setB(b); // "a" knows about "b"
em.persist(a);
```

This is perfect for our purpose, since we can build the complete tree and then mark its root as persistent. This will store the entire tree. As for object identity, Hibernate objects have identity within the program, but this is the case in which it is not a problem. Identity of objects is destroyed after the program exits, but the whole tree is automatically loaded from the backend database once the program starts again and the data is requested. Objects in this run have different "pointers", but the tree is still valid, because each object has its own abstract ID in the backend database. This ID is used to rebuild identity of objects and, based on it, the tree as well. Optimizations of object-oriented databases based on object-relational mapping are often a problem because they add a large amount of overhead, as discussed before. This is partially compensated in Hibernate by possibility of configuration and also possibility of finetuning the underlying database backend. Finally, there are also some issues when using JPA and object-relational mapping. The issue, although only a small one, is that application designer needs to do the design of data structures in SQL as if doing design for relational data model. That is certainly more complicated than defining data structures in object-relational databases. On the other hand, available code generators solve this issue for programmers.

To sum up and answer questions from section 5: both hierarchical and flat DOM representation is possible. Collections and object references are at programmer disposal in the same range as the programming language offers. That makes design challenges like M:N mapping rather simple. Maps are usually a part of the language, therefore simple textual representation of both attributes and style information is possible. Relation between VIPS blocks can be represented by a collection of `Relation` objects. Similar design can be used for Burget's grid.

6 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 for storage.

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.

The second half then outlined basic features of two database approaches convenient for storing the data identified in the first part. 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 many shortcomings which would need to be compensated by additional program code. Therefore it won't be the first choice in most cases.

References

1. 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
2. Hors, A. L., Hégaret, P. L., Nicol, G., Wood, L., Champion, M., Byrne, S.: Document Object Model (DOM) Level 3 Document Object Model Core. W3C Recommendation. April 2004
3. Stenback, J., Hégaret, P. L., Hors, A. L.: Document Object Model (DOM) Level 2 Document Object Model HTML. W3C Recommendation. January 2003
4. W3C: Document Object Model (DOM) Technical Reports (overview)
5. 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.
6. 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.
7. Burget, R.: Layout Based Information Extraction from HTML Documents. The Ninth International Conference on Document Analysis and Recognition. 2007.
8. Tai, K. C.: The tree-to-tree correction problem. J. ACM 26(3). 1979
9. 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.
10. EJB 3.0 Expert Group: JSR 220: Enterprise JavaBeans^{TM}, Version 3.0; Java Persistence API. May 2006
11. Silberschatz, A., Korth, H. F., Sudarshan, S.: Database System Concepts, 5th edition. New York: McGraw-Hill.