Spatiotemporal Annotation Graph (STAG): A Data Model for Composite Digital Objects

Smriti Yamini
Department of CSE
University of California San Diego
smriti@sdsc.edu

Amarnath Gupta
San Diego Supercomputer Center
University of California San Diego
gupta@sdsc.edu

Abstract

In this demonstration, we present a database over complex documents, which, in addition to a structured text content, also has update information, annotations, and embedded objects. We propose a new data model called Spatiotemporal Annotation Graphs (STAG) for a database of composite digital objects and present a system that shows a query language to efficiently and effectively query such database. The particular application to be demonstrated is a database over annotated MS Word and PowerPoint presentations with embedded multimedia objects.

1. Introduction

The current demonstration is motivated by our on-going projects with the National Archives and Records Administration (NARA) of USA, whose missions include long-term preservation of now-digital government records and creating means to search them.

We consider the problem of performing content-based search on annotated composite digital documents stored in a digital library. A composite digital document, produced by popular software like MS Word, Adobe Acrobat and MS PowerPoint, is a document which in addition to a (semi-) structured textual content also has (i) multi-author document update information (ii) annotations (e.g., comments) made over the primary document content, (iii) embedded objects like images, audio and video, some of which can be, in turn, complex digital documents themselves, and (iv) in some cases, like PowerPoint documents, the presentation order of the document components. Today's operating environments allow us to search through collections of digital objects by textual search, and potentially, using structural search using an XML encoding of the documents. We believe that a digital library should enable a user to search over the structure and content of the entire composite document, and contend that XML by itself is adequate to capture the semantics of these composite documents.

We posit three observations to claim the inadequacy of XML as a data model for composite documents:

- Let us assume a document is structured like a tree. An annotation is like marking up a portion of this tree with a different color. Further, annotations can be nested (like a comment on a comment), and typed (from different authors). If we model both the base document together with the annotation structure as a single tree (i.e., XML), the tree would be deep and very complex. So it will be very difficult to formulate queries on the document itself, the annotation by itself, and on both. A similar observation led [4] to propose the multi-colored tree model.

- While XML provides a structure to a document, the structure itself does not capture any semantics. Specifically, it is not designed to capture the presentation aspect of the documents. Thus, the order of appearance of embedded media objects is not captured by the XML structure. L.Sheng et al discuss multimedia presentation graphs in [7]. Further, if the media objects are themselves structured (like MPEG-4 for video), a standard XML representation would not bring up the domain-specific semantics of the embedded document.

- A direct XML-based representation of a document cannot capture updates made to the document – while it is possible to add time-stamps as node attributes, we cannot represent both document order as well as temporal order through the same set of nodes in a tree structure. [6] presented a model for representing update-able XML documents on the web. If the data is spatio-temporally varying, such as in an animation, it cannot be represented at all.

The goal of this demonstration is to present a new data model called Spatiotemporal Annotation Graph (STAG) that is based on a directed acyclic graph. We have developed a set of algebraic operations over STAGs. We show

that the STAG model captures all aspects of composite documents discussed above and can be queried through an extension of SQL.

2. Our Data model

Intuitively, a STAG can be thought of as a confluence of three graphs defined over the same set of nodes and edges – the first is a *data graph* whose nodes are atomic data elements and whose edges represent the structural relationships between them; the second graph represents the *presentational relationships* among them; and the third adorns subgraphs of the data graph with different colors, where the colors stand for *annotation assignments* made by different parties. In this model, we assume the documents not to be self-referential, which makes the data graph acyclic. By construction, the presentation and the coloring components of the graph do not produce any cycles, thus making the whole STAG acyclic. The model of STAG is an extension of the Annotation Graphs [1] developed for speech databases.

Figure 1 shows an example of a STAG. The round nodes belong to the data graph, while the square nodes are the annotation nodes. Annotations *C1* and *C3* are made by the same user and hence have the same color, while annotation *C2* is made by a different user.

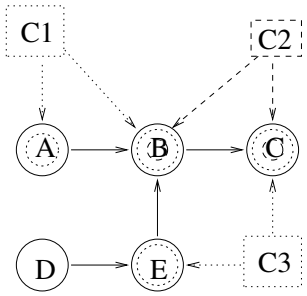

Figure 1. Example of a STAG

Definition 1 *A STAG is a weakly connected directed acyclic graph G consisting of a set of nodes V, node-types $\tau(V)$, node colors $\gamma(V)$, a set of edges E, edge-types $\tau'(E)$, and edge labels $\lambda(E)$, such that the following properties hold:*

STAG0: γ_{0i} is a distinguished family of colors called the *base colors* – every node and edge of the *i*-th STAG has the color γ_{0i}, and no other STAG has the same base color.
STAG1: Every node has a type assigned to it from type names $\tau(V)$. A hierarchy may be defined over node types, allowing inheritance.

STAG2: A τ'_0 edge between two nodes indicate structural containment in the sense of XML documents. The τ'_0 children of a node in a STAG are generally unordered. However, every node has a null-able attribute called `l_order` that may specify the *local order* of the node with respect to its parent. A node's children will either be unordered, or be locally ordered – partial local ordering is disallowed.
STAG3: Every node in the graph is **spatiotemporal** in the sense that it has a *time-stamp* of creation/modification, and a null-able *location* attribute. Two nodes with non-null location values may be connected through an edge of type *spatial* having a label like `contains` or `overlaps`. Similarly, a *temporal edge* having a label like `after` may connect two nodes, indicating the child node is presented after the parent. Sibling nodes that are temporal children of the same parent overlap in time for presentation.
STAG4: Every node inherits the γ_{0_i} color from its parent node. However, any color other than the *base* color is inherited by a node from another node if there is an incoming *spatial* edge from that node.

2.1. How to add annotations?

This section gives a brief description about how a user can annotate a STAG. For example, we have a PowerPoint presentation. A user may want to annotate some slide or some part of the slide for his reference or as a note for anybody who uses those slides. Assume that the presentation is of lecture notes for Distributed Systems. Now slides on "Concurrency Control" can also be used for Distributed Databases. So a user annotates certain text and images in some of the slides which can be used for distributed databases with appropriate remarks. Then by simply querying on these remarks we can extract all the material which is common to both the subjects.

A user can create an annotation by using the following query:

```
create annotation as <annotation> on
<object>;
```

The `object` here is specified by an *object id* and it refers to a subgraph/node. The object id of the *object* can be obtained by an appropriate query. When a user adds an annotation, a new annotation node is created having a color which is specific to the user and an edge is added from the newly created node to all the target nodes. And these target nodes inherit the color of the annotation node.

Similarly, to delete an annotation we have ,

```
delete annotation <string>; or
delete annotation <object>;
```

3. Algebra over STAG

The algebra required for manipulation of a STAG is a blend of operators required for tree operations, graph operations and extended versions of relational operations.

The *color* operator returns the color of the edge or node specified as an argument to it. The operators *parent/child* give the parent/child of the given node. The operations *lastchild, firstchild, RightSibling, LeftSibling, Ancestor and Descendant* give the node sharing the respective relationship with the input node.

The *Union* works like a set-union, same for *intersection*. The operators *Difference, Projection and Selection* are also defined for a STAG. By projecting a STAG over a given color we can extract the graph of a document or annotations made by a particular user. The grouping and aggregate operators have the same meaning as their relational counterparts ,the difference being that in this case they operate over edges and nodes. The temporal operators like *next, follows, parallel and overlaps* determine whether a node is *next* to another or if a node *follows* another or if it *overlaps* another. *Parallel* is equivalent to an exact overlap.

4. Demonstration

We will demonstrate a STAG database populated with everyday digital documents such as emails, Word files and PowerPoint presentations each of which may have other embedded objects. An email, for instance, will have a PowerPoint presentation which will further embed images and audio.

We will first demonstrate how document as the above can be converted into a STAG instance, and then show how a collection of such STAG instances can be queried, retrieved and visualized using an SQL-like query language over our algebra.

We are currently evaluating STAG database implemented on PSE [5], a persistent storage system from Object Store against XXL [2]. We have a command line interface where a user may invoke various commands to perform tasks like insert a new STAG, annotate a STAG, query the STAG database, etc. The interface consists of mainly three components : the document converter, the query processor and, the visualizer and document viewer. The Figure 2 shows a the architecture of the system.

4.1. Document Converter

First we need a STAG instance of the user's document. This is obtained using an executable called *convert2stag*. It is invoked as:

$$convert2stag[-d] - f < inputfile > [-o < outputfile >]$$

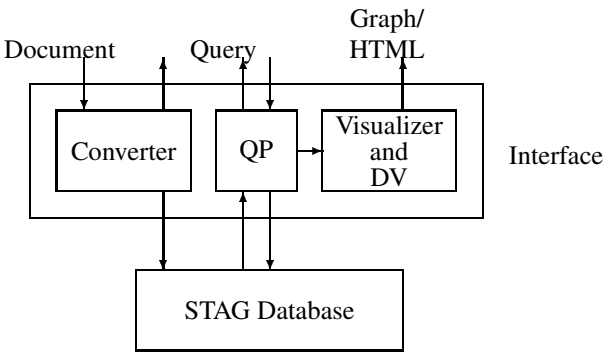

Figure 2. Architecture of the System

It has various command line inputs some of which are : *d*- to display the STAG produced, *f*- for the input file name and *o*- for the output file name. If the latter is not provided then it uses the name of the input file itself. The input file name should contain the absolute path of the file if the file is not in the current directory and the right extension. If the extension of the file does not match the file format, the system tries to detect the file format and confirms with the user. If the detected file format is correct it continues to process the file based on it instead of the file extension. Otherwise, it exits. The output file has a *.stag* extension. Next,we add the newly created STAG to the database. For this we have a command *insertstag* which inserts a STAG from a file into the database. The syntax for the command is:

```
insertstag from <file> ;
```

The $< file >$ must be a *.stag* file as created in the previous step.

The functionality of *convert2stag* and *insertstag* can be coupled into a single command for the ease of the user as a user *must* execute these two steps in order to insert the document into the STAG database. The system described above is a command line interface. It can be extended further to a graphical user interface where the user can browse the directory structure and select the input file and execute the above steps at the click of a button.

4.2. Query Processor

Once we have populated the database, we would want to perform different tasks on it like query or annotate STAGs. Towards this end we have a SQL-like query language that would enable a user to query as well as annotate the STAGs in the database. There are commands to create/delete/modify annotations, query annotations by user/document, to list all the STAGs in the database and

the documents they represent, etc. We will demonstrate queries like the following:

1. Find the PowerPoint presentation slide where there is an animation and image overlapping a text box.

```
select s from PPTGraph
where s.label = 'slide' and
descendant(s) = (t,i,a) and
t.mediaType = 'Text' and
i.mediaType = 'Image' and
a.mediaType = 'animation' and
(i overlaps t) and (a overlaps
t)
```

2. Find the PowerPoint presentation which contains a sequence of three slides having the same title and the slide after these has only text.

```
select p from PPTGraph
where p.docType = 'ppt' and
child(p) =
sequence(s1,s2,s3,s4) and
s1.label = s2.label = s3.label
= s4.label = 'Slide' and
s1.title = s2.title = s3.title
and
childCount(s4) =
childCount(s4,'Text')
```

   The function *sequence* specifies that p has (locally) ordered children s_1, s_2, s_3 and s_4.

3. Find the e-mail containing a word document that has some section title with the word "Demo", at least two images and the second last modifying author was "ag".

```
select e from emails
where e.docType = 'e-mail' and
child(e) = w and
w.docType = 'MSWord' and
descendant(w) = s and
s.label = 'SectionTitle' and
substr(s.value,'Demo') and
w.imageCount() > 2 and
getChild(reverseSort(descendant(w),
'annotation','modification
time'),2).author = 'ag'
```

4.3. Visualizer and Document Viewer

By default, the STAG environment is in the textual mode. So the result of any query or annotation is displayed in the text format that is a graph/subgraph is given by its adjacency list. When a user wants to annotate a subgraph(s)/node(s) he/she has to first select it through a query. Thus on annotation of a STAG, first the selected subgraph(s)/node(s) is displayed and then the annotated version of it. However, it is most convenient for potential users to use a graphical mode where they can *view* the STAGs directly. This is called the Visualizer. The user can change from the text mode to the visual mode by simply invoking the command *visual*. Henceforth, the results of all the queries and annotations will be displayed graphically. For drawing graphs we use the graph drawing software GraphViz[3].

Along with the selected subgraph we also display its neighborhood. This neighborhood is computed by determining the k-nearest neighbors of the subgraph where k is a function of the average height of the tree.

The Document Viewer provides the user with the option of viewing the changes on the document itself. He/She can view the original document, document with all the changes or the document with some selected changes where the selection is done based on the criteria specified by the user. User can select certain annotations based on *objectid /userid /*query. The *objectid* refers to the id assigned to every annotation and *userid* refers to the id of the user who added the annotations.Irrespective of the format of the document, every document would be displayed in the HTML format. The annotations may be highlighted, underlined, quoted or within boxes.

References

[1] S. Bird and M. Liberman. Annotation graphs as a framework for multidimensional linguistic data analysis. In *Towards Standards and Tools for Discourse Tagging: Proceedings of the Workshop*, pages 1–10. Association for Computational Linguistics, Somerset, New Jersey, 1999.

[2] M. Cammert, C. Heinz, J. Krmer, M. Schneider, and B. Seeger. A Status Report on XXL - A Software Infrastructure for Efficient Query Processing. *Data Engineering Bulletin*, 26(2):12–18, 2003.

[3] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz - An open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.

[4] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful XML: One hierarchy isn't enough. In *Proc. of the ACM SIGMOD Conference*, 2004.

[5] G. Landis, C. Lamb, T. Blackman, S. Haradhvala, M. Noyes, and D. Weinreb. ObjectStore PSE: a Persistent Storage Engine for Java. In *Proc. of the 2nd Int. Workshop on Persistence and Java(tm) (PJW2)*, 1997.

[6] K. Norvag. Temporal query operators in XML databases. In *Proceedings of the ACM Symposium on Applied Computing*, pages 402–406, 2002.

[7] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, pages 572–581, 1999.