EXTENDING DATABASES TO SUPPORT IMAGE EDITING

Greg Speegle, Allen M. Gao, Shaowen Hu

Department of Computer Science
Baylor University
Waco, TX. 76798
speegle@cs.baylor.edu

Le Gruenwald

School of Computer Science
University of Oklahoma
Norman, OK 73019
gruenwal@cs.ou.edu

ABSTRACT

In order to understand similarity between images, recent research has focused on adaptable searches [9] and fuzzy queries [4]. However, one of the best means for determining similarity between images is to know how the image was created [2]. If the image is a combination of other images in the database, then there is a great deal of similarity between the base images and the created one. This requires extending the database to support image editing operations. We have built a preliminary system which does this by using a web-based image editor and a image server. The editor and the server understand a *logical model language* that represents images. This paper then explores the issues of performance for deriving images from a sequence of operations.

1. INTRODUCTION

We have built a prototype of an object-oriented multimedia database management system (MMDBMS). Our MMDBMS stores the editing operations, collectively called *transforms*, used to create media objects, instead of the objects themselves. This provides three advantages. First, since the editing operations require 170 bytes of space on average, this method significantly reduces storage requirements for already compressed data [10]. Second, the transforms can be used to improve similarity search by providing semantic information about the image [2]. Third, the history of the development of an image can be captured in the database.

However, our MMDBMS requires support by multimedia editing tools in order to be successful. Since different editors use different operations, a mapping from the operations performed to a common editing language is needed. Our system uses the *logical model language*, denoted LML, which captures the basic image editing functions. A web-based image editor called Ritet translates user operations into LML operations. The collection of all LML operations which are used to create an imaged are stored by the image server as a *transform*. When Ritet requests an image that is stored as a transform, the instantiation subroutine creates the image from the list of operations and delivers it to the editor. Thus, our system provides all of the basic services of an image server with support for image editing.

Ideas similar to our work can be found in [3] and [8]. In [3], multimedia objects are treated as binary large objects (BLOBs). Editing of media objects is translated into operations on BLOBs. Handles to BLOBs are stored in a relational database. Unfortunately, BLOB operations are not sufficient to perform all required media object operations. For example, there is no way to describe the dithering needed to double the size of an image. In [8], a notion similar to transforms is presented. There, scripts are passed to image editing tools in order to create new images. However, their approach performs these operations outside the database itself. This can lead to problems with deleted references and inefficient specifications, both of which can be solved by including the transforms within the database.

The rest of the paper is organized as follows. Section 2 presents the LML and our meta-structure. Section 3 presents implementation details for creating images from lists of operations. Section 4 concludes this work and outlines the rest of our project. It should be noted that this paper focuses on images, but we will apply this work to other media types.

2. THE MODEL

The database must have a mechanism to support image editing. This mechanism controls insertion and deletion of transforms and images in the database, and creates images from transforms. This requires a meta-structure for the data and a language for modeling image editing operations. The meta-structure consists of a *transform-dag* which represents the history of the transform. An arc in a transform-dag from o_1 to o_2 means that o_1 is used to create o_2. In this case, o_1 is called the base image, and o_2 is called the derived image. As its name suggests, a transform-dag is a directed acyclic graph. The roots of every transform-dag are stored objects.

The arcs in the transform-dag represent changes made to one image in order to create another. Those changes are

$$\begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta)+y_r(\sin\theta) \\ \sin\theta & \cos\theta & y_r(1-\cos\theta)-x_r(\sin\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 1: The Mutate Matrix for Θ^o counter-clockwise rotation

$$\begin{bmatrix} 1 & 0 & X \\ 0 & 1 & Y \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2: The Mutate Matrix to move all pixels X to the right and up Y

enumerated in a specification. The specification consists of a sequence of simple operations which capture all of the editing commands performed by users. Thus, for this work we use five operations–Merge, Define, Mutate, Modify and Combine. These operations form a natural set of basic operations for image manipulations in systems which do not add information to images by techniques such as drawing. Such a "nothing gets added" system would prohibit the user from drawing a bird into a picture, but it would allow copying a picture of a bird from one image and putting it into another. Adding information, manipulating the color palette and channel operations are needed in a complete set of image editing tools. Finally, this restricted model is designed only for RGB (red-green-blue) color models. For simplicity, we present the Mutate operation in detail. The other operations are defined in [10].

The Mutate operation is used to alter the form of an image or a region of an image. The mutation is accomplished by mapping an image into a form where every pixel is a 5-tuple, (x,y,r,g,b), where (x,y) is the coordinate location of the pixel and (r,g,b) is the color component. Each (x,y) component is put into a 3 x 1 vector (where the last element in the vector is usually 1). This vector is then multiplied by a 3x3 matrix in order to get a desired effect. An example of a Θ^o counter-clockwise rotation is in Figure 1. Mutate operations can also move a region to a new location within the image (Figure 2), or change the size of the image (Figure 3). Note that in the case of increasing the size of the image, there are pixels which have no color value. Interpolation is needed to fill in the missing values. Likewise, non-integer coordinates require interpolation in order to get the pixel values for the integer coordinates. The resulting image is then mapped back to pixel format, with the order of the pixels defined by their location in the file. Note the examples presented here assume the origin is in the lower lefthand corner. Slightly different matrices are used for images where the origin is in the upper lefthand corner.

A challenge in shape recognition is invariance under rotation, sizing and other differences which can be captured by the Mutate operation [7]. Clearly, by knowing that the image is a rotation, or size change of a known shape, it is obvious that the shape is maintained. Similar benefits related to other problems can be solved by knowing the derivation of the image [2].

$$\begin{bmatrix} X & 0 & 0 \\ 0 & Y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 3: The Mutate Matrix to increase the size by factor of X height and Y width

3. IMPLEMENTATION ISSUES

Each of the basic operations has been implemented to determine the time required to perform the operation and the space required to store the specification. We simplified the implementation by restricting defined regions to rectangles. Experiments were carried out on a lightly loaded dual processor Pentium II running at 300 MHz with 512MB RAM running Solaris 2.6. In order to simulate the overhead of a database system, each specification entry was stored as a separate file which had to be opened and read for each operation. The image files were all JPEG images. The implementation uses a Java applet front-end called Ritet (pronounced "write it") which stands for Remote Image Transform Editing Tool [6]. This Java applet supports simple editing operations on images and translates them into the LML. The LML operations are stored on a server.

The backend server manages the specifications and instantiates the images on demand. It is composed of two parts, DBProxy and DBServer. DBProxy handles communication with Ritet and DBServer, including graceful termination on timeouts. The use of DBProxy allows us to update either the server or Ritet without interfering with the other component. We have implemented DBProxy in Java and DBServer in C++. DBServer instantiates the specifications. The current implementation simply writes the created image to a file, and then sends the URL to Ritet.

The instantiation code for each operation has been completed. The Define operation allocates the memory needed for the selected region, and copies the pixels into the locations. All other operations are then performed within this memory until another Define operations is performed. The most complex operation on a single image is Mutate. There are four cases for the Mutate operation. The easiest case simply moves the relative location of the defined region.

Operation	Msec	Relative
Define	0.00189	1.00
Modify	0.00042	0.22
Mutate	0.00570	3.02
Combine	0.00372	1.97

Figure 4: Time of LML Operations

This requires virtually no time. Case 2 involves pasting the defined region back into the picture. If no additional space is needed in the picture, this requires only the time to modify memory values. Case 3 occurs when the new region extends beyond the bounds of the current image. In this case, the image size is increased to hold the new pixels. Note this is not always done in image editing software, but this prevents loss of data values. The allocation of memory causes case 3 to be up to 5 times slower than case 2. The fourth case for the Mutate operation is a rotation. If the rotation is not 90 degrees, then additional memory must be allocated. The pixels in this new memory are set to "transparent" so that they will not be seen if pasted back into the original picture. Combinations of Mutate operations can be found by multiplying the mutation matrices.

In order to develop a formula to estimate the processing cost of instantiating images, the operations are applied to various images ranging from 100 x 100 to 900 x 900, with a comparable percentage of each image selected as the defined region. Figure 4 contains the actual and relative speeds of the operations in msec per pixel. Each operation is part of a longer specification which was performed ten times. In all cases, times farthest from the mean were eliminated until the standard deviation was less than 5% of the mean. No more than 2 samples were eliminated by this approach, and all were clearly "outliers" caused by system loading or other external factors. The means were plotted against the defined region size, and the data indicated highly linear relationships. The coefficients were generated using the method of least squares.

Figure 4 indicates a nearly linear relationship between the time to perform the operations (e.g., a Mutate operation takes 3 times as long as a Define). This allows a simple mechanism to tune the database with respect to storing either the transform or the actual image. A simple regression test can determine the time to perform the Define operation for a system. Once that is known, for any specification, the total time required for instantiation can be approximated with the formula

$$T = \sum_{i=0}^{k} (1 + 2 * C_i + 3 * M_i + 0.2 * F_i) * P_i * S \quad (1)$$

where T is the total time for the operation, i is the number of the defined regions, C_i is the number of Combine operations on that region, M_i is the number of Mutate operations on the region, F_i is the number of Modify operations on the region, P_i is the number of pixels in the region, and S is the time to perform a Define operation for a region of 1 pixel.

For example, let $S = 0.00189$, as found in Figure 4. Assume the transformation consists of defining a region of 1833 bytes. Let the region be modified to more green, then blurred (Combine), then rotated. Next, assume a region of 93,810 bytes is defined. This region is made more red, then sharpened (Combine) The estimated time for instantiation would be

$$6.2 * 1833 * 0.00189 + 3.2 * 93,180 * 0.00189 = 585 \quad (2)$$

By experimentation, the actual time to perform this transformation is 596 msec. The simple approximation can be used to determine if the time to dynamically instantiate the image is acceptable, or if the image should be instantiated off-line.

One operation omitted from Figure 4 is Merge. The Merge operation requires a source image to be instantiated and then pasted into the target image. The paste operation is exactly the same as the Mutate case 2 and 3 mentioned before (in fact, the exact same code is used). The duration of the Merge operation depends on the time required to instantiate the source, and to a lesser extent, on the time required to paste in the image. There is also a small amount of overhead time needed to find the base image required by the specification for the transform. Thus, each Merge operation is approximately as complex as an entire specification without Merges. To determine if a transform with a Merge operation should be dynamically instantiated or stored in the database as an image, the time required to instantiate the Merge must be added to the time required to perform the other operations.

Finally, before an image can be returned to the user, the system must read the base images from the database. If the images are compressed, they must be decompressed. The time required to read and decompress images is not included in the instantiation estimates. Our current implementation also recompresses the data before sending the information to the editing tool. We are working on methods to eliminate this additional time delay.

4. CONCLUSION AND CURRENT STATUS

By extending databases to handle image editing operations, it is possible to store a sequence of editing operations in the database instead of the image itself. Sequences save space, maintain a history of images and provide additional information for similarity search. We have defined a common language to be used by image editing tools in order to capture these operations. This paper explores the implemen-

tation issues related to the time for instantiation of images. Every multimedia database system will have different requirements with respect to the time required to instantiate images. For example, a studio which does touch-ups on images will not need high performance, but will store very large, high-resolution images. Such a system would be best tuned to have very high thresholds for the time to instantiate an image. On the other hand, an online museum would have few changes (perhaps none) and would have very low thresholds for instantiating images. Figure 4 indicates a linear relationship exists between the time to define a region and the time to perform operations on it. This allows simple database tuning to determine if an image or a transform should be sorted in the database. Merge operations require as much time as instantiating an image.

The MMDBMS is now based on the Postgres object-relational database system [11]. Base images are stored as large objects within the database and derived images are stored as simple tuples. The specifications are stored in tables with a foreign key linking them to the derived images. DBProxy and Ritet are unchanged. The image instantiation routine is an external routine, and the images are still passed as URLs to the editor. New experiments are underway to determine if the linear relationship between operations holds in the database environment. An automated feature extraction routine has been added as a trigger to the MMDBMS. The trigger is fired whenever a new derived image is saved in the database. Open issues related to implementation include optimization [12] and extended the LML to eliminate the restrictions noted in Section 2.

Despite the presence of an automated feature extraction routine, it is still an open question as how to best use the specifications to perform similarity search. For example, certain characteristics of shape similarity, such as rotation and scale invariance, are difficult in traditional automatic feature extraction [7], but are easy when it is known that the shape is simply part of a Mutate operation [1]. However, it is also possible to use the semantic information contained in the view-dag to help with the k-nearest neighbor problem [2], but the appropriate metrics are currently unknown.

Finally, the work done on images should be extended to other, larger media sources such as audio and video. The MPEG-7 [5] initiative is an attempt to standardize the operations which can be performed on a video object. If successful, this can lead to a LML for video. Once that is established, a meta-structure very similar to the one presented here can be used to provide direct database support for video editing.

5. REFERENCES

[1] Mike Aars. Automatic feature extraction for edited images. Master's thesis, Baylor University, 1999.

[2] L. Brown, L. Gruwnwald, and G. Speegle. Issues in using specifications to improve content-based search of multimedia data. In *1999 International Symposium on Database Applications in Non-Traditional Environments*, pages 195–202, November 1999.

[3] J. Cheng, N. Mattos, D. Chamberlin, and L. DeMiciel. Extending relational database technology for new applications. *IBM Systems Journal*, 33(2):264–279, 1994.

[4] R. Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–9, June 1998.

[5] International Organization for Standaridization. Mpeg-7:context and objectives. ISO/IEC/JTC1/SC29/WG11, 1998. Available on www.cselt.it/mpeg/standards/mpeg-7/mpeg-7.htm.

[6] Minghui Gao. Technical manual for ritet, a remote image transform editing tool for a mulimedia object-oriented database server. Master's thesis, Baylor University, 1999.

[7] G. Lu and A. Sajjanhar. Region-bassed shape representation and similarity measure suitable for content-based image retireval. *Multimedia Systems*, 7(2):165–174, 1999.

[8] G. Schloss and M. Winblatt. Building temporal structures in a layered multimedia data model. *Proceedings of ACM Multimedia 94*, pages 271–278, October 1994.

[9] T. Seidl and H.-P. Kriegel. Efficient user-apatbale similiarity search in large multimedia databases. In *Proceedings of the Twenty-third International Conference on Very Large Databases*, pages 506–515, August 1997.

[10] G. Speegle, X. Wang, and L. Gruenwald. A meta-structure for supporting multimedia editing in object-oriented databases. *Lecture Notes in Computer Science*, 1405:89–102, July 1998. Proceedings of the 16th British National Conference on Databases.

[11] Ting Zhou. Automatic image feature extraction in a multimedia database system. Master's thesis, Baylor Univeristy, 2000.

[12] Harold Zou. Technical manual for optimizing image specifications. Master's thesis, Baylor University, 1998.