Image database management systems design considerations algorithms and architecture
Nes, N.J.

Citation for published version (APA):
Nes, N. J. (2000). Image database management systems design considerations algorithms and architecture

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

Database Assisted Image Processing

The activities in image and database research fields seem opposites of a spectrum. Image processing usually involves object (image) at a time processing and database systems use set at a time processing. However, taking a closer look, they are more related than one might expect. In this chapter we will demonstrate a database approach for image processing, which will open new ways to optimize image processing algorithms against large image collections.

3.1 Data Structures

Storage consideration has long been driving the design of image processing packages. Packages, such as SCILIMAGE, Horus[112], IUE, Khoros, Matlab and PhotoShop, store images in two dimensional arrays of pixel values. This simplified data representation requires no storage for the spatial component of a pixel; its location is implicit. The implicit spatial component is used throughout the algorithms.

A usual further reduction is obtained by using limited pixel value types. For example, 255 gray levels present in an image can be stored in a single byte pixel value. Although the storage requirement drops, it also creates an overflow problem. Performing a pixel value operation can result in an overflow, i.e. the value does not fit in the byte representation.

The storage considerations for modern image processing packages are less import. With memory prices dropping quickly, current workstations easily hold 256M of memory, which is more than adequate for image processing geared towards a limited set of fully exploded images (1-7MB a piece).

Although the two dimensional array approach has its advantages, it also has some difficulties.
• Optimization decisions are visible to the user

• Can not handle arbitrary shaped images

• Low level application programmers interface (API) only, i.e. only pixel level operations are used.

• Cannot handle other regular or ir-regular grids, for the spatial component.

An important database concept is to have a generally defined type at the logical level and to hide the storage optimizations at the physical level. Such a general definition for the logical level is defined in the Image Algebra, which defines an image as a mapping from a spatial domain $X$ into a range value domain $F$.

In the Monet DBMS we would map an image to a $\text{BAT}[X,F]$. For such mapping we need atomic types to represent the pixel positions and values. The atoms currently provided in the Monet image database system are shown in Table 3.1 together with the MEL packages supporting these types. These packages can be extended with operators needed for new image operations.

<table>
<thead>
<tr>
<th>PIXEL Type Mel Package</th>
<th>Implementation Type representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single value types</td>
<td></td>
</tr>
<tr>
<td>monochrome</td>
<td>bit</td>
</tr>
<tr>
<td>grayscale</td>
<td>byte</td>
</tr>
<tr>
<td>real valued</td>
<td>float</td>
</tr>
<tr>
<td>2D Vector types</td>
<td></td>
</tr>
<tr>
<td>location</td>
<td>int</td>
</tr>
<tr>
<td>gradient</td>
<td>float</td>
</tr>
<tr>
<td>3D Vector types</td>
<td></td>
</tr>
<tr>
<td>color</td>
<td>rgb</td>
</tr>
<tr>
<td>color</td>
<td>HSI</td>
</tr>
</tbody>
</table>

Table 3.1: Pixel Types

This logical image definition solves the earlier mentioned problems related to the two dimensional array approach. It supports arbitrary shaped images, and can handle different spatial representations. The costs for this flexibility is high, because we potentially loose the storage savings of the implicit spatial component. Later in this Chapter we will come back to this storage overhead.
3.2. **PRIMITIVES**

The data models for images and database relations are closely related, but can we also show that their primitives are closely related? In this investigation we follow the operation classes as defined by the image algebra (Section 2.1.1).

Image restriction on its range values can be mapped directly on the well known select operations in database systems. For example, a restriction to all pixels with values above a constant $k$ maps to a selection of all records with attribute range value above $k$. Restrictions on its spatial domain map to a range selection (when the spatial ranges are known). When the restricting spatial set is known a natural-semi-join operation can be used. Let $a,b$ be a $bat(X,F)$ and $k \in F$ then formally the mapping is as follows:

$$a|>_{k} \leftrightarrow a.select(k, nil)$$

$$a|b \leftrightarrow a.semijoin(b)$$

The image extension primitive maps on a combination of the set union and difference operations. An extension of $a$ with $b$ is formally mapped as follows:

$$a|b \leftrightarrow a.union(b, kdiff(a))$$

The range($a$) and domain($a$) operations map to the project operation in Monet, projecting the column of interest. They are mapped as follows:

$$\text{range}(a) \leftrightarrow ['nil' \sim a]$$

$$\text{domain}(a) \leftrightarrow [a \sim 'nil']$$

Induced image operations in Horus[112] map onto combinations of natural joins and scan operations in Monet. A binary image operation can be mapped using a natural join between the two spatial attributes and a scan over the resulting table, performing the binary pixel operation on the range value attribute. In Monet the induced image operation $\lambda$ between two image $a$ and $b$ can be concisely expressed as follows:

$$a\lambda b \leftrightarrow [\lambda](a, b)$$

Monet already has some global reduction operations, namely: min, max, sum, count and histogram. They perform the obvious reduction operations on BATs. Since Monet has no general BAT aggregation operation, each global reduction operation requires an implementation effort. This can be done both by MIL procedures and C functions. A general interface for such operations can be defined to reduce this effort. This interface specifies three operations: the init, next and finalize operations. The init function initializes the reduction operation, e.g. setting variables. The next operation is called
for each element in the BAT. The finalize operation is called wrap up the result.

Spatial operations are somewhat more complex to map into MIL. The problem with such mapping is that the spatial domain of the result should be a priori known. Then it is similar to the induce image operations. First, a scan is performed to transform the spatial domain of the result into the spatial domain of the original image, calling $f$ for each position. The function $f$ should be defined for the two dimensional space. Finally, a join is required to look up the range values. Some $y \in \text{domain}(f)$ may require values outside the domain of $a$, i.e. $f(y) \notin X$. These will not be present in the induced image.

$$a \circ f \leftrightarrow [f](\text{domain}(f)).\text{join}(a)$$

To check whether the template image operations maps onto database operations, we first need a mapping of a template. A template is defined as an image of images, which maps to a table of tables. A template operation will map to a scan of the template pixels, which are again images. For each template image, an induced image operation is performed. The resulting image, which will after being reduced using an image reduce operation, form the resulting pixel value.

$$t \otimes a \leftrightarrow |\text{template}_{\text{op}}(t, [t \sim \text{const} a])|$$

where

$$\Lambda(\bigcirc(t_y, a)) = \text{template}_{\text{op}}(t_y, a)$$

The $[t \sim \text{const} a]$ construction creates a temporary template from the image $a$, so both operands of template_{op} have the same table of tables format. The template_{op} operation performs the real induced image operation and reduction.

To make the mapping of template operations to the BAT algebra operations clear we will explain the mapping of a well known image processing operation, convolution. The image convolution is a template operation which requires an induced image multiplication between the image and each template image. The resulting images are reduced using a summation image reduction operation. Figure 3.2 shows the implementation of image convolution in MIL.

As shown the image algebra operations map onto the binary relational algebra operations. Since templates are just a special kind of images, their operations also map into the algebra operations.

### 3.3 Benefits of BAT representation

The mapping solution proposed solves the identified problems with the two dimensional array representation. The true benefits should still be made
3.3. BENEFITS OF BAT REPRESENTATION

3.3.1 Image Integration

We simulated the Horus image representation and operations with the core of the Monet database system. An important immediate benefit is the availability of index structures which come with the binary tables. These index structures can be used to improve the performance of some image operations drastically. For example, searching the spatial and range domains can be optimized using proper accelerators[8].

Segmentation algorithms cluster pixels to form segments. For example [47] describes a segmentation algorithm based on k-means clustering in color
space. Having an index structure or automated lookup table on the pixel values is beneficial.

### 3.3.2 Simplification of Data Structures

The binary table is Monet's main data structure. The decision for a single complex structure (storing relatively simple atomic types) has proven to be crucial for its core development and its impressive performance. With the mapping we proofed (again) that this is also a powerful data structure to handle image data types. The single structure can be used as image, but also as data structure to store derived data sets. This means users (i.e. image researchers) only have to understand a single complex data type, which significantly decreases the learning curve.

The reuse of the BAT data structure has another important advantage, namely code reuse. All image algebra operations map on combinations of existing relational operators. There is no additional implementation effort. Introducing new image processing operations can easily be done by supplying the pixel value operations required for it. In many cases a simple MIL procedure suffices.

### 3.3.3 Query Optimization

The key to fast responds in a DBMS is the query optimizer. By mapping images into tables we can benefit from these techniques, i.e. the query optimizer has all information to choose an optimal query execution plan. Some techniques are introduce shortly.

**Translation invariant templates**

A possible optimization is to use the properties of a translation invariant template. When we know that a template is translation invariant, we can select one image from the template, for example $t_y$ with $y = (0, 0)$, and use it to represent the template. All other template images $t_x$ can be regenerated using this image and a translation of its pixel values from $y$ to $x$.

To illustrate, instead of doing an induced image operation followed by a reduction for each position of the template $t$, we translate the original image $a$ over $x$ for each $x \in X$, where $X$ is the position set of the template representant image. For each resulting image perform a binary scalar induced image operation, where the scalar is the pixel value at position $x$. The last step is to reduce the set of images using the reduce operation as the operator of an induced image operation. Using this scheme reduces the number of lookups of the pixel values of the template images. The steps are displayed graphically in Figure 3.3.3.
3.3. **BENEFITS OF BAT REPRESENTATION**

Reuse of intermediate results

Having translation invariant templates and some values \( v \) in the image \( t_y \) exists multiple times, we can further reduce the number of operations required. For each value in \( t_y \) we apply the induced image operation, i.e. removing any duplicates. The optimized set of operations is displayed graphically in Figure 3.3.3

For some translation invariant templates, the values of the image \( t_y \) are not used. They are set to the unit value, i.e. it only expresses the selection of the image pixel values. For example in a uniform filter the values are not used, only the positions are of interest. The image algebra denotes these as *neighborhood operators*. These operators only require the translation and reduction steps of the template image product. Example neighborhood operations are uniform filter, median filter and dilation and erosion.

Further optimizations, such as "sliding windows", for the uniform image filter can be supported by specialized operations. These operators place a window over the original image, calculate a single result pixel from using the pixels in the window, slide the window, and calculate the next pixel reusing the pixel calculations in the intersection of the two windows. There should be a way to express when a image template operation should use these operations, i.e. a translation invariant template where all template value are equal can use the fast uniform image filter[61] implementation.

Figure 3.3.3 shows the implementation of an optimized translation invariant convolution. The translation invariant template is represented by a
single BAT[Y,F]. First the unique values are discovered. Using these the image multiplication is done. Each resulting image is translated over the given vector. The resulting images are combined (added) using the sum_images operation. The resulting values are divided by the sum of the pixel values in the translation invariant template.

```plaintext
proc image_mul(BAT im, BAT ty) := {
    return [*](im,ty);
}

proc sum_images(BAT im) := {
    var res := im.fetch(0);
    var rest := im.slice(1,im.count());
    rest@batloop() {
        res := [+](res,$t);
    }
    return res;
}

proc image_ti_convolution(BAT[any,any] image, BAT[any,any] template) := {
    var unique_values := template.reverse().kunique();
    unique_values :=
        unique_values.join(unique_values.reverse);
    var mul_ims :=
        [image_mul]([unique_values const im],unique_values);
    var trans_ims := [translate](mul_ims, template.reverse() );
    var sum_ims := sum_images(trans_ims);
    return sum_ims;
}
```

Figure 3.4: Translation Invariant Convolution

Filter Decomposition

Some translation invariant templates can be decomposed into a set of smaller templates. Such decomposed templates reduce the number of operations needed to calculate a image template product. For example a 3x3 template is decomposed into a 1x3 and a 3x1 template the number of operations required for the template image product reduces from 9 to 6 per pixel. So from $O(n^2)$ to $O(n)$.

A query optimizer is the right place to find out such decompositions and use it to optimize the query plan.
3.3.4 Parallelism

Mapping the image algebra operations onto relational algebra operations opens a road to parallel execution. For relational algebra operations many parallel algorithms exist. For example a template image product can be performed in parallel. The work needed for all the induced image operations can be spread over the pool of processors. Even lower granularity parallelism can be achieved using horizontal decomposition parallelism. A table is horizontally decomposed into multiple smaller fragments. These fragments are distributed over the processors and the work is done there. The resulting fragments are on return gathered to form the result.

An other form of parallelism is single instruction multiple data (SIMD), which can be found in nearly all modern CPU. Example instruction sets are Intel’s MMX and SSE, AMD’s 3D Now, Motorola’s Altivec, and Sun VIS. All are geared at multi-media applications, but these operations can just as easily be used by database systems. Having the image algebra mapped on the relational algebra it will seamlessly use the SIMD optimized operations. In this thesis, parallelism is not considered.

3.3.5 Performance and Storage Improvements

The mapping of images into Monet’s BATs can be implemented with a storage overhead comparable to the two-dimensional array image definition commonly used by the image processing software packages. In this section we also indicate how to further optimize storage requirements.

To understand the solution, we first explain the BATs data structure. Figure 3.3.5 shows a typical BAT structure used in Monet. Each binary table consists of a Binary unit (BUN) heap, to store the head and tail of the relation. Each column has a fixed or variable type and optionally multiple search accelerators. Fixed sized atoms are stored directly in the BUN heap. Variable sized atoms are stored in a separate heap. In the BUN heap the position of the variable atom is stored. A BAT with a head type oid and a tail type chr will require 8 bytes, because integers require 4 bytes alignment on most systems.

Although this storage scheme proved flexible, deployment in the data mining showed another way to reduce the storage requirements. Let us take a look at Figure 1, which shows the decomposition of a relational table into BATs. The head of the BATs contain enumerations of unique object identifiers. This information can be represented by a single object identifier, indicating the first value, and a counter. Leading to virtual object identifiers (voids){12}. Using the void type reduces the storage requirements drastically, since only one column needs to be stored.

We can use the virtual object identifiers (voids) trick to solve the redundant spatial information. Just using the void type as the head is not
sufficient, because the spatial information would be lost. Therefore, we introduce a separate BAT to store dimensionality information, i.e. the image width and height. Image operations using the spatial component should take care of handling these images based on void BATs. They should lookup the dimensions and generate the implicit spatial component before performing the actual operation.

This way, we achieve a huge storage reduction for the spatial component $X$ of image. Can we also save storage for the $F$ component? A 1024 by 1024 24 bit color image requires, after spatial reduction, still leaves $1024 \times 1024 \times 3$ bytes, i.e. 3 MB, which is still huge when considering image databases with over 1M images (i.e. 3 TB databases). Fortunately, the cardinality of the different values in the images is usually much lower than the number of
pixels. This fact is also exploited by image compression schemes, such as found in the JPEG image format. In our Monet implementation a reduction may be possible using an indirection to find the pixel value. Instead of storing all values $f$ directly, a position in a lookup table is stored. \footnote{A reduction from 3 bytes to 1 byte can be achieved when less than 256 different pixel values exist. Many images coming from the world wide web have this property.}

An important consequence of the lookup table is the possibility to defer non-spatial operations on the pixel values to the lookup table, reducing the number of operations dramatically. Example candidate image operations are unary and binary-scalar induced operations.

### 3.4 Experiments

To demonstrate that this approach is also feasible from a performance point of view, we performed some experiments with one of the most important image operations, the image convolution. As a sanity check, we compare our implementation against the implementations done in Horus.

Horus is a new image process library developed by the university of Amsterdam. It is designed to be a general image processing library, intended for image analyzing tasks, implemented in C++ making heavy use of code templates. The main focus of the library is performance. Therefore, some of the generality of the image algebra is given up in favor of processing speed. The Horus library only looks at translation invariant templates, called kernels. These are the ones used most frequently. Also the image implementation of Horus is focused on square shaped images, it lacks implicit image segment support.

To show whether our optimizations have the desired effects we compare our default convolution of invariant templates with a convolution using the indirected pixel value, i.e., using a lookup tables "ColorMap" representation which requires less multiplication operations.

We compared the execution times of image convolution operations for various image sizes, from 16x16 to 512x512 and various templates. The results for each template are shown in the figures 3.4, 3.4, 3.4 and 3.4. These figures show the results for the Horus convolution and both Monet convolutions with and without pixel value lookup table optimization.

Figure 3.4 shows that the mapping of images to BATs is only approximately 20% more expensive. This is relative small since it is achieved by the existing relational operators. The use of the lookup table directly pays off it gives a performance gain off 20% over Horus.

Figure 3.4 shows that both Monet implementations have approximately the same performance, which is 25% better than Horus. The reason is that the Horus implementation can not make use of the fact that all values in the
template are equal. A user could solve in Horus by calling the neighborhood operation, but that shows the optimization to the user.

Figure 3.4 shows that the mapping pays off. The Horus cannot handle arbitrary shaped images and templates and therefore the convolution implementation has to go through the whole 3x3 template values, even though 4 values are 'zero'.

The last figure shows the combination of the previous two optimizations, i.e. reuse the intermediates and no calculations for the 'zero' template values. This gives already about 50% performance increase.

### 3.5 Requirements

The requirements coming from using binary tables as the data structure for images are:

- DBMS should be extensible with new abstract data types, for pixel
3.6. CONCLUSIONS

\[
\begin{array}{ccc}
  y-1 & y & y+1 \\
  y+1 & 1 & \\
  y & 2 & 3 & 4 \\
  y-1 & 5 & \\
\end{array}
\]

Figure 3.8: Execution times of convolution operation

\[
\begin{array}{ccc}
  y-1 & y & y+1 \\
  y+1 & 1 & \\
  y & 1 & 1 & 1 \\
  y-1 & 1 & \\
\end{array}
\]

Figure 3.9: Execution times of convolution operation

position and pixel values.

- The DBMS requires a void data type with virtual identifiers.
- The DBMS requires BATs which store values using a lookup table.
- The DBMS should allow for operator overloading.

In our research we found a huge gap between the research communities of image processing and database management systems. A long history of different single object versus set at a time processing has widened the gap. With this chapter we hope to reduce this gap a little, since we know both worlds can benefit substantially from each other.

3.6 Conclusions

In this chapter we showed our mapping of images to BATs, i.e. binary tables. We indicated how a default implementation of the image algebra operations
can be achieved. This also proves the completeness of our approach. Using this representation we indicated many roads towards optimization. We indicated how these optimizations are obtained by the query optimizer to find better query plans.

We showed that sets of images can be compressed with an additional BAT interface. The interface allows for transparent access to BATs with compressed data.