1 TWO DATA ORGANIZATIONS FOR STORING SYMBOLIC IMAGES IN A RELATIONAL DATABASE SYSTEM

Aya Soffer and Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Science
University of Maryland at College Park
College Park, Maryland 20742

aya@umiacs.umd.edu and hjs@umiacs.umd.edu

Abstract: A method is presented for integrating images into the framework of a conventional database management system (DBMS). It is applicable to a class of images termed *symbolic images* in which the set of objects that may appear are known a priori. The geometric shapes of the objects are relatively primitive and they convey symbolic information. Both the pattern recognition and indexing aspects of the problem are addressed. The emphasis is on extracting both contextual and spatial information from the raw images. A logical image representation that preserves this information is defined. Methods for storing and indexing logical images as tuples in a relation are presented. Indices are constructed for both the contextual and the spatial data, thereby enabling efficient retrieval of images based on contextual as well as spatial specifications. Two different data organizations (integrated and partitioned) for storing logical images in relational tables are proposed. They differ in the way that the logical images are stored. Sample queries and execution plans to respond to these queries are described for both organizations. Analytical cost analyses of these execution plans are given.

INTRODUCTION

Images (or pictures) serve as an integral part in many computer applications. Examples of such applications include CAD/CAM (computer aided design and manufacturing) software, document processing, medical imaging, GIS (geographic information systems), computer vision systems, office automation systems, etc. All of these applications store various types of images and require some means of managing them. The field of *image databases* deals with this problem [8]. One of the major requirements of an image database system is the ability to retrieve images based on queries that describe the content of the required image(s), termed *retrieval by content*. An example query is “find all images containing camping sites within 3 miles of fishing sites”.

In order to support retrieval by content, the images should be interpreted to some degree when they are inserted into the database. This process is referred to as converting an image from a *physical* representation to a *logical* representation. The logical representation may be a textual description of the image, a list of objects found in the image, a collection of features describing the objects in the image, a hierarchical description of the image, etc. It is desirable that the logical representation also preserve the spatial information inherent in the image (i.e., the spatial relation between the objects found in the image). We refer to the information regarding the objects found in an image as *contextual information*, and to the information regarding the spatial relation between these objects as *spatial information*. Both the logical and the physical representation of the image are usually stored in the database. An index mechanism based on the logical representation can then be used to retrieve images based on both contextual and spatial information in an efficient way.

There are many image database systems (e.g., Virage [18], QBIC [11], Photobook [13], FINDIT [17] as well as others [2, 5, 6, 12]). Most systems treat the image as a whole, and index the images based mainly on color and texture. A few systems try to recognize individual objects in an image. These systems do not, however, address the issues of spatial relationship between the objects. Other systems deal with indexing tagged images (images in which the objects have already been recognized and associated with their semantic meaning) in order to support retrieval by image content.

In our work, we have chosen to focus on images where the set of objects that may appear are known a priori. In addition, the geometric shapes of these objects are relatively primitive and they convey symbolic information. Our application is the map domain where many graphical symbols are used to indicate the location of various sites such as hospitals, post offices, recreation areas, scenic areas etc. We call this class of images *symbolic images*. Other similar terms found in the literature are *graphical documents*, *technical documents*, and *line drawings*. Limiting ourselves to symbolic images simplifies object recognition enabling using well-known methods in document processing.

In this paper, we present methods for integrating symbolic images into a conventional database management system (DBMS). In our application, we make use of a relational DBMS although our ideas are applicable to other DBMS’s. These methods offer solutions for both the pattern recognition and indexing aspects of the problem. We describe how to incorporate the results of these methods into an existing spatial database based on the relational model. Our emphasis is on extracting both contextual and spatial information from the raw images. The logical image representation that we define preserves this information. The logical images are stored as tuples in a relation. Indices are constructed on both the contextual and the spatial data, thus enabling efficient retrieval of images based on contextual as well as spatial specifications. It is our view that an image database must be able to process queries that have both contextual and spatial specifications, in addition to any traditional query.

We propose two different data organizations, termed *integrated* and *partitioned*, for storing images in relational tables. They differ in how logical images are stored. All of the examples and experiments in this paper are from the map domain. However, images from many other interesting applications fall into the category of symbolic images. These include CAD/CAM, engineering drawings, floor plans, and more.

The main contribution of this work lies in demonstrating how a traditional DBMS can be used to store and retrieve images and how partitioning this data effects the performance of the database. While the database and pattern recognition techniques that we use are well-known, the novelty of this work is in adapting and integrating these techniques into one system that provides a comprehensive solution for storing and retrieving images in a DBMS. We suggest solutions for all of the steps that are involved in this integration. These steps include: image acquisition, interpretation, storage, indexing, and retrieval. The main issues that need to be resolved are:

1. finding an image interpretation procedure whose results can be stored as entries in a traditional database in such a way that both the contextual and spatial information inherent in the image will be preserved.
2. what data organization is most suitable for the types of queries that are common in this application.
3. determining what strategies to use when computing answers to queries (i.e., how to use the double indexing on both contextual and spatial data efficiently).
4. finding ways to compute their costs.

The rest of this paper is organized as follows. We first present definitions as well as the notation used. Next, we outline the image input system used to convert images from their physical representation to their logical representation as they are input to the database. We continue by describing how images are stored in a database management system using the two data organizations that we propose including schema definitions and example relations. This is followed by sample queries along with execution plans and cost estimates for these plans. We conclude with some observations as well as directions for future research.

DEFINITIONS AND NOTATIONS

Below we define some terms and the notation used in the remainder of the paper. A *general image* is a two-dimensional array of picture elements (termed *pixels*) $p_0, p_1, \ldots, p_n$. A *binary image* is a general image where each pixel has one of two possible values (usually 0 and 1). One value is considered the foreground and the other the background. A general image is converted into a binary image by means of a threshold operation. A *symbol* is a group of connected pixels that together have some common semantic meaning. In a given application, symbols will be divided into *valid symbols* and *invalid symbols*. A *valid symbol* is a symbol whose semantic meaning is relevant in the given application. An *invalid symbol* is a symbol whose semantic meaning is irrelevant in the given application. A *class* is a group of symbols all of which have the same semantic meaning. All invalid symbols belong to a special class called the *undefined class*.

A *symbolic image* is a general image I for which the following conditions hold: 1) Each foreground pixel p_i in I belongs to some symbol. 2) The set of possible classes $C_1, C_2, \ldots, C_n$ for the application is finite and is known a priori. 3) Each symbol belongs to some class. 4) There exists a function f which when given a symbol s and a class C returns a value between 0 and 1 indicating the certainty that s belongs to C.

Images can be represented in one of two ways. In the *physical image* representation, an image is represented by a two-dimensional array of pixel values. The physical representation of an image is denoted by I_{phys}. In the *logical image* representation, an image I is represented by a list of tuples, one for each symbol $s \in I$. The tuples are of the form: $(C, certainty, (x, y))$ where $C \neq$*undefined*, (x, y) is the location of s in I, and $0 < certainty \leq 1$ indicates the certainty that $s \in C$.

IMAGE INPUT

Conversion of input images from their physical to their logical representation is performed using methods common in document analysis [9]. These methods use various pattern recognition techniques that assign a physical object or an event to one of several pre-specified classes. Patterns are recognized based on some features or measurements made on the pattern. A library of features and their classifications, termed the *training set library*, is used to assign candidate classifications to an input pattern according to some distance metric. Each candidate classification is given a certainty value that approximates the certainty of the correctness of this classification.

We have adapted these methods to solve the problem of converting symbolic images from a physical to logical representation. Figure 1.1 is a block diagram of the image input system that we have developed for this purpose. It is driven by the symbolic information conveyed by the image. That is, rather than trying to interpret everything in the image, it looks for those symbols that are known to be of importance to the application. Any other symbol found in the image is labeled as belonging to the undefined class. This system is described in detail in [15]. In this paper we show how to integrate this system into a DBMS, thus we only give a short overview of the image input system here. A symbolic image I_{phys} is input to the system in its physical representation. It is converted into a logical image by classifying each symbol s found in I_{phys} using the training set library. An initial training set library is constructed by giving the system one example symbol for each class that may be present in the application. In the map domain, the legend of the map may be used for this purpose.

The system may work in two modes. In *user verification mode*, users verify the classifications before being input to the database. The training set is modified to reflect the corrections that the user made for erroneous classifications. In *automatic mode,* classifications are generated by the system and input directly to the database. The user determines the mode in which the system operates. In general, the system should operate in user verification mode until the recognition rate achieved is deemed adequate. Then, the system can continue to process the input images automatically.

The output of applying the conversion process to I_{phys} is a logical image where the tuples are of the form $(C, certainty, (x, y))$ where $C \neq$ *undefined*, $0 <$ *certainty*

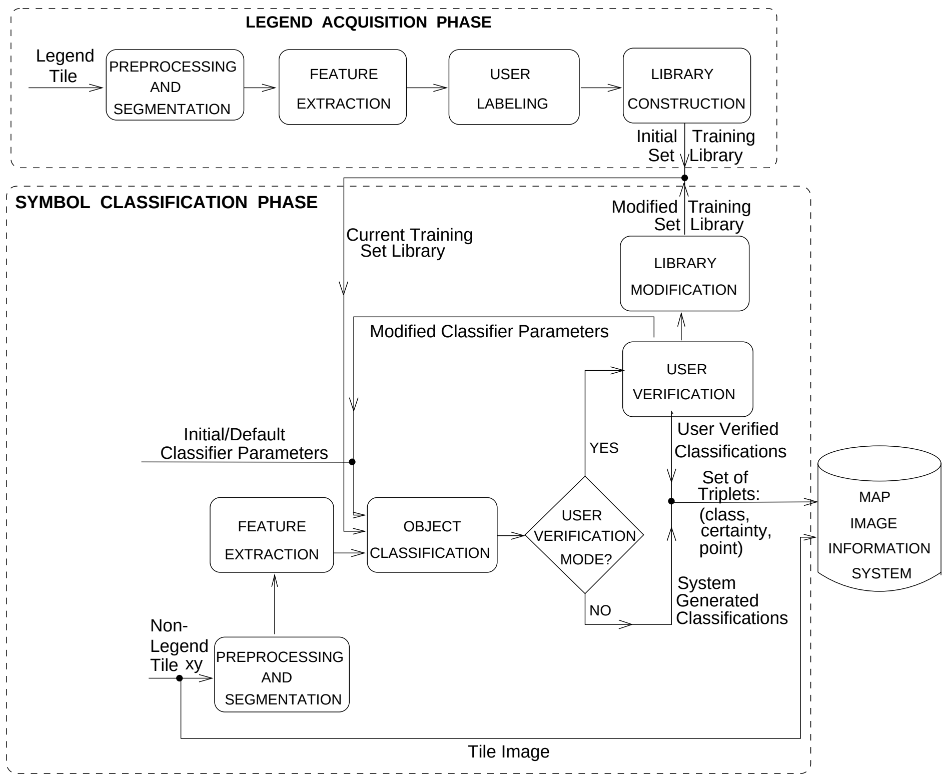
LEGEND ACQUISITION PHASE
Legend Tile
PREPROCESSING AND SEGMENTATION
FEATURE EXTRACTION
USER LABELING
LIBRARY CONSTRUCTION
Initial Training Set Library
SYMBOL CLASSIFICATION PHASE
Modified Training Set Library
Current Training Set Library
LIBRARY MODIFICATION
Modified Classifier Parameters
USER VERIFICATION
Initial/Default Classifier Parameters
YES
User Verified Classifications
USER VERIFICATION MODE?
Set of Triplets: (class, certainty, point)
MAP IMAGE INFORMATION SYSTEM
FEATURE EXTRACTION
OBJECT CLASSIFICATION
NO
System Generated Classifications
Non-Legend Tile xy
PREPROCESSING AND SEGMENTATION
Tile Image

Figure 1.1 Image input system

≤ 1 indicating the certainty that $s \in C$, and (x, y) is the location of s in I_{phys}. For each image, a set of such tuples is inserted into a spatial database as described in the following section. In addition, the raw image I_{phys} (i.e., the image in its physical representation) is also stored.

IMAGE STORAGE

Images and other information pertaining to the application are stored in relational tables. The database system that we use for this purpose is SAND [1, 3] (denoting spatial and non-spatial database), developed at the University of Maryland. It is a home-grown extension to a relational database, in which the tuples may correspond to geometric entities such as points, lines, polygons, etc. having attributes which may be both of a locational (i.e., spatial) and a non-locational nature. Both types of attributes may be designated as indices of the relation. For indices built on locational attributes, SAND makes use of suitable spatial data structures. Attributes of type image are used to store physical images. Query processing and optimization is performed following the same guidelines of relational databases extended with a suitable cost model for accessing spatial indices and performing spatial operations.

We propose two different data organizations for storing the images in relational tables. They differ in the way logical images are stored. In the *integrated organization*, all tuples of the logical images are stored in one relation. In the *partitioned organization*, the tuples are partitioned into separate relations resulting in a one-to-one correspondence between relations and classes of the application. For example, tuples $(C, certainty, (x, y))$ of a logical image for which $C = C_1$ are stored in a relation corresponding to C_1. The motivation for the partitioned organization is that many queries in in an application using symbolic images need to access all symbols that are assigned the same classification. The part of the query that selects all tuples that belong to the same classification is repeated each time such a query is posed. The partitioned organization makes this repetitive selection at query time unnecessary by providing the option to partition the logical images relation. The partitioned organization is only suitable for applications in which the number of classes is relatively small, as there is one relation for each class and a proliferation of relations would make the database too complex. In the case of symbolic images, this is a reasonable assumption. The number of different symbols used to convey symbolic information (which corresponds to the number of classes) will most likely not be very large, otherwise it would be hard to keep track of or look up the semantic information that is conveyed by each symbol. For example, in the map domain this information must be contained in the legend of the map which is limited in space. Hence, the partitioned organization seems to be reasonable for a database that stores symbolic images. The partitioned organization also enables efficient use of spatial indices while processing spatial queries by using a spatial join operator (e.g., [14]).

Integrated Organization

```
(CREATE TABLE classes              (CREATE TABLE physical_images
  name STRING PRIMARY KEY,            img_id INTEGER PRIMARY KEY,
  semant STRING,                      descriptor STRING,
  bitmap IMAGE);                      upper_left POINT,
                                      raw IMAGE);
(CREATE TABLE logical_images
  img_id INTEGER REFERENCES physical_images(img_id),
  class STRING REFERENCES classes(name),
  certainty FLOAT (CHECK certainty BETWEEN 0 AND 1),
  loc POINT,
  PRIMARY KEY (img_id,class,loc));
```

Figure 1.2 Schemas for the relations `classes`, `physical_images`, and `logical_images`.

The schema definitions given in Figure 1.2 define the relations in the integrated organization. We use an SQL-like syntax. The `classes` relation has one tuple for each possible class in the application. The `name` field stores the name of the class (e.g., star), the `semant` field stores the semantic meaning of the class in this application (e.g., site of interest). The `bitmap` field stores a bitmap of an instance of a symbol

class	semantics	bitmap
S	harbor	Ⓢ
square	hotel	◙
scenic	scenic view	↷
T	customs	Ⓣ
R	restaurant	Ⓡ
P	post office	Ⓟ
M	museum	Ⓜ
K	cafe	Ⓚ
waves	beach	⊜
triangle	camping site	▲
B	filling station	Ⓑ
arrow	holiday camp	⬆
cross	first aid station	⊕
fish	fishing site	🐟
H	service station	Ⓗ
inf	tourist information	ℹ
pi	picnic site	π
air	airfield	✈
star	site of interest	✱
telephone	public telephone	☎
box	youth hostel	■
U	sports institution	Ⓤ

Figure 1.3 Example instance for `classes` relation.

image_id	descriptor	raw	upper_left
image_1	tile 003.012 of Finnish road map	Fig. 1.5	(6144,1536)
image_2	tile 003.013 of Finnish road map	Fig. 1.6	(6656,1536)

Figure 1.4 Example instance for `physical_images` relation.

representing this class. It is an attribute of type `IMAGE`. The `classes` relation is populated using the same data that is used to create the initial training set for the image input system (i.e., one example symbol for each class that may be present in the application along with its name and semantic meaning). See Figure 1.3 for an example instance of the `classes` relation in the map domain.

The `physical_images` relation has one tuple per image I in the database. The `img_id` field is an integer identifier given to the image I when it is inserted into the database. The `descriptor` field stores an alphanumeric description of the image I that the user gives when inserting I (this is meta-data). The `raw` field stores the actual image I in its physical representation. It is an attribute of type `IMAGE`. The `upper_left`

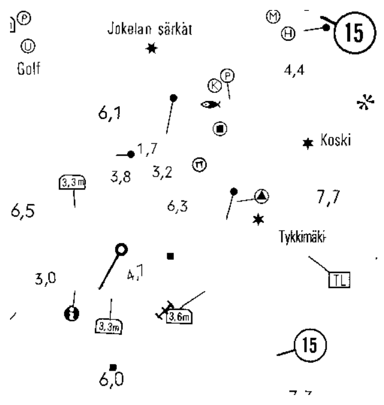

Figure 1.5 Example: image_1.

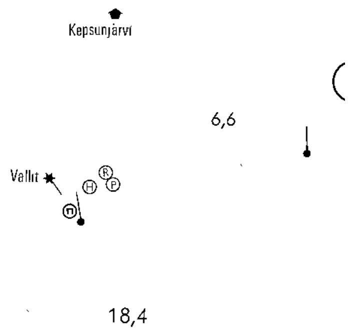

Figure 1.6 Example: image_2.

field stores an offset value that locates the upper left corner of image I with respect to the upper left corner of some larger image J. This is useful when a large image J is tiled, as in our example map domain. Subtracting this offset value from the absolute location of s in the the non-tiled image J yields the location of s in the tile I that contains it. It is an attribute of type POINT. Any additional meta-data that the user may wish to store about the images such as how they were formed, camera angles, scale, etc. can be added as fields of this relation. See Figure 1.4 for an example instance of the physical_images relation in the map domain.

The logical_images relation stores the logical representation of the images. It has one tuple for each candidate class output by the image input system for each valid symbol s in each image I. The tuple has four fields. The img_id field is the integer identifier given to I when it was inserted into the database. It is a foreign key referencing the img_id field of the tuple representing I in the physical_images relation. The class and certainty fields store the name of the class C to which the image input system classified s and the certainty that $s \in C$. The loc field stores the (x, y) coordinate values of the center of gravity of s relative to the non-tiled image. See Figure 1.7 for an example instance of the logical_images relation in the map domain for the images given in Figures 1.5 and 1.6.

Constructing Indices Indices are defined on the schemas defined above as follows (in SQL-like notation):

```
CREATE INDEX cl_sem ON classes (semant);
CREATE INDEX cl_name ON classes (name);
CREATE INDEX pi_id ON physical_images (img_id);
CREATE INDEX pi_ul ON physical_images (upper_left);
CREATE INDEX li_cl ON logical_images (class certainty);
CREATE INDEX li_loc ON logical_images (loc);
```

image_id	class	certainty	location
image_1	M	1	(6493,1544)
image_1	P	0.99	(6161,1546)
image_1	H	0.99	(6513,1566)
image_1	U	1	(6167,1583)
image_1	star	0.99	(6332,1586)
image_1	P	0.99	(6432,1622)
image_1	K	1	(6416,1636)
image_1	fish	1	(6411,1661)
image_1	scenic	0.99	(6630,1662)
image_1	square	1	(6422,1693)
image_1	star	0.99	(6540,1712)
image_1	pi	0.99	(6396,1741)
image_1	triangle	1	(6475,1784)
image_1	star	1	(6474,1814)
image_1	cross	0.79	(6291,1854)
image_1	box	0.74	(6357,1862)
image_1	inf	1	(6226,1937)
image_1	box	1	(6280,2011)
image_2	arrow	0.99	(6861,1544)
image_2	scenic	0.72	(6803,1565)
image_2	pi	0.99	(6849,1756)
image_2	R	0.71	(6849,1756)
image_2	P	0.99	(6858,1771)
image_2	H	0.99	(6827,1775)
image_2	U	0.79	(6827,1775)
image_2	pi	0.99	(6800,1807)
image_2	R	0.99	(6800,1807)

Figure 1.7 Example instance for the `logical_images` relation in the map domain. The tuples correspond to the symbols in the images of Figures 1.5 and 1.6.

`cl_sem` and `cl_name` are alphanumeric indices. They are used to search the `classes` relation by `semant` and `name`, respectively. The `pi_id` index is also alphanumeric. It is used to search the `physical_images` relation by `img_id`. `pi_ul` is a spatial index on points. It is used to search the `physical_images` relation by the coordinates of the upper left corner of the images. `li_cl` is an alphanumeric index. It is used to search the `logical_images` relation by `class`. It has a secondary index on attribute `certainty`. Thus, tuples that have the same class name are ordered by certainty value within this index. `li_loc` is a spatial index on points. It is used to search the `logical_images` relation by location (i.e., to deal with spatial queries regarding the locations of the symbols in the images such as distance and range queries). The spatial indices are implemented using a PMR quadtree for points [10].

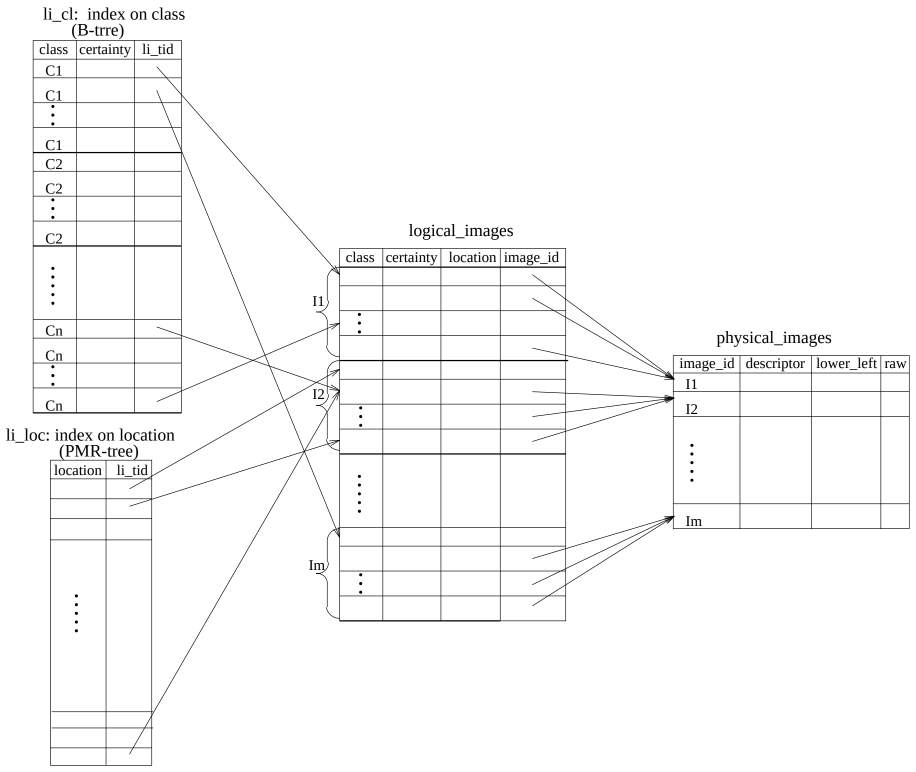

Figure 1.8 File structures for logical and physical images using the integrated organization.

Observe that the file structures resulting from the integrated organization are very similar to the file structures used by inverted file methods for storing text [4]. An inverted file consists of two structures. A vocabulary list which is a sorted list of words found in the documents, and a posting file indicating for each word the list of documents that contain it and information regarding its position in the document. The vocabulary list is actually an index on the posting file, and is used to locate the record of the posting file corresponding to a given word on disk. In our organization, the `logical_images` relation corresponds to the posting file. The index `li_cl` on this relation plays the role of the vocabulary list. The main difference from text is that as we are dealing with 2-dimensional information rather than 1-dimensional information, we need more elaborate methods to store and index the locational information. In particular, just storing the location, as is done for text data, is insufficient. In order to answer spatial queries efficiently, these locations must be sorted by use of a spatial index. Figure 1.8 illustrates the file structures used following the integrated organization that correspond to similar file structures used for text data.

Partitioned Organization

In the partitioned organization, tuples are partitioned into separate relations resulting in a one-to-one correspondence between relations and classes of the application. For example, tuples $(C, certainty, (x, y))$ of a logical image for which $C = C_1$ are stored in a relation corresponding to C_1. Figure 1.9 gives schema definitions for relations of the partitioned organization corresponding to the `logical_images` relation of the integrated organization. Both the `classes` and `physical_images` definitions are identical to those in the integrated organization. The only difference between the organizations is the way the logical images are stored. In the partitioned organization, there is one relation, `cl_part` for each class *cl* in the application. Each relation `cl_part` contains the logical images tuples $(C, certainty, (x, y))$ for which $C = cl$. This is equivalent to the result of a selection operation: `SELECT FROM logical_images WHERE class` $= cl$. See Figure 1.10 for example instances of relations `star_part`, `P_part`, `scenic_part`, and `pi_part` for the images in Figures 1.5 and 1.6.

```
for each class cl in application
    (CREATE TABLE cl_part
     img_id INTEGER REFERENCES physical_images(img_id),
     certainty FLOAT (CHECK certainty BETWEEN 0 AND 1),
     loc POINT,
     PRIMARY KEY (img_id,loc));
```

Figure 1.9 Schemas for the `cl_part` relations in the partitioned organization.

Constructing Indices Indices are defined on the separate class schemas of the partitioned organization as follows (in SQL-like notation):

```
for each class cl in application
    CREATE INDEX cl_cert ON cl_part (certainty);
    CREATE INDEX cl_loc ON cl_part (loc);
```

Each instance of the `cl_part` relation has an alphanumeric index on `certainty` and a spatial index on `loc`. The spatial index is used to deal with queries of the type “find all images with sites of interest within 10 miles of a picnic area” by means of a spatial join operator. Figure 1.11 illustrates the file structures for the partitioned organization corresponding to file structures used for text data.

RETRIEVING IMAGES BY CONTENT

As mentioned above, we distinguish between contextual information and spatial information found in images. Similarly, we distinguish between query specifications that are purely contextual and those that also contain spatial conditions. A *contextual specification* defines the images to be retrieved in terms of their contextual information (i.e., the objects found in the image). For example, suppose we want to find all images that contain fishing sites or campgrounds. A *spatial specification* further constrains the star_part:

image_id	certainty	location
image_1	0.99	(6332,1586)
image_1	0.99	(6540,1712)
image_1	1	(6474,1814)

scenic_part:

image_id	certainty	location
image_1	0.99	(6630,1662)
image_2	0.72	(6803,1565)

P_part:

image_id	certainty	location
image_1	0.99	(6161,1546)
image_1	0.99	(6432,1622)
image_2	0.99	(6858,1771)

pi_part:

image_id	certainty	location
image_1	0.99	(6395,1741)
image_2	0.99	(6849,1756)
image_2	0.99	(6800,1807)

Figure 1.10 Example instances of relations `star_part`, `scenic_part`, `P_part`, and `pi_part`. The tuples correspond to the symbols in the images of Figures 1.5 and 1.6.

required images by adding conditions regarding spatial information (i.e., the spatial relations between the objects).

In order to describe the methods that we use for retrieving images by content, we first present some example queries. Next, we demonstrate the strategies used to process these queries. We conclude by analyzing the expected costs of these strategies (termed *plans*) and compare the data organizations (i.e., integrated and partitioned).

Example Queries

The example queries in this section are first specified using natural language. This is followed by two equivalent SQL-like queries. The first assumes an integrated organization and the second assumes a partitioned organization.

Query Q1: display all images containing a scenic view .

```
display PI.raw
```

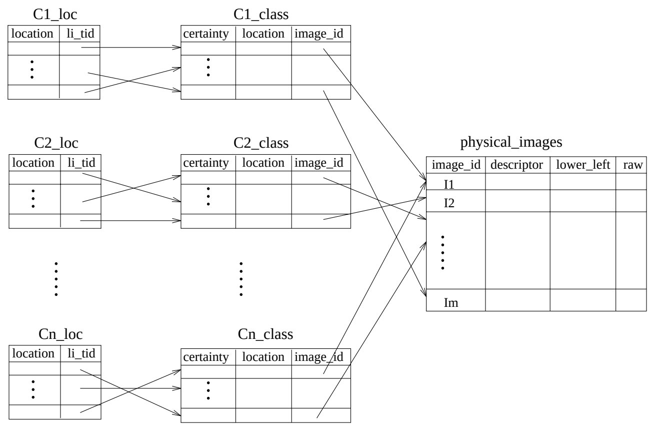

Figure 1.11 File structures for logical and physical mages using the partitioned organization.

```
    from logical_images LI, classes C, physical_images PI
    where C.semantics = "scenic view" and C.name = LI.class
        and LI.image_id = PI.image_id;

display PI.raw
    from scenic_part SC, physical_images PI
    where SC.image_id = PI.image_id;
```

Notice that in order to write SQL-like queries for the partitioned organization, the names of the relations corresponding to each partition must be known. This can easily be overcome by having the system assign names to these relations. These names are derived from the `class` attribute of relation `classes`. Two functions that perform this name conversion are provided. `get_rel_name` returns the name of a relation given the class name. `get_class` returns the class name given a relation name. Thus, there is no need for the user to know the names assigned by the system to these relations.

Query Q2: display all images containing a scenic view within 5 miles of a picnic site.

```
display PI.raw
    from logical_images LI1, logical_images LI2, classes C1,
        classes C2,  physical_images PI
    where C1.semantics = "scenic view"
        and C2.semantics = "picnic site"
        and C1.name = LI1.class and C2.name = LI2.class
        and distance(LI1.location,LI2.location) < 5
        and LI1.image_id = LI2.image_id
        and LI1.image_id = PI.image_id;

display PI.raw
```

```
from scenic_part SC, pi_part PIC, physical_images PI
where distance(SC.location,PIC.location) < 5
    and SC.image_id = PIC.image_id
    and PIC.image_id = PI.image_id;
```

The function `distance` takes two geometric objects (e.g., two points) and returns a floating point number representing the Euclidean distance between them.

Query Processing

The following plans outline how responses to queries Q1 and Q2 are computed using the two data organizations. These plans utilize the indexing structures available for each organization. Indices on alphanumeric attributes are capable of locating the closest value greater than or equal to a given string or number. Indices on spatial attributes are capable of returning the items in increasing order of their distance from a given point (this is termed an *incremental nearest neighbor operation*) [7]. This operation may optionally receive a maximum distance, D, and it will stop when the distance to the next nearest neighbor is greater than D. Thus, it returns all neighbors within D of a query point in increasing distance. Direct addressing of a tuple within a relation is possible by means of a tuple identifier (or *tid* for short). All index structures have an implicit attribute that stores this tid. The X^{th} plan, labeled Px_I, uses the integrated organization. The X^{th} plan, labeled Px_P, uses the partitioned organization.

Query Q1: display all images containing a scenic view.

Plan P1$_I$: Search using an alphanumeric index on `class`.

```
Get all tuples of **logical_images** which correspond to **"scenic view"**
    (use index **li_cl**)
For each such tuple **t**
  display the physical image corresponding to **t**
```

Plan P1$_P$ Search the **scenic view** partition sequentially

```
For each tuple **t** of the **"scenic view"** partition
  display the physical image corresponding to **t**
```

Query Q2: display all images containing a scenic view within 5 miles of a picnic site. Finding a suitable plan for query Q2 gives rise to many query optimization issues. Most of these issues are also applicable to spatial databases (e.g., [1]). To see the complexity of these issues, we give two different plans for computing an answer to query Q2 using each organization. The first uses only alphanumeric indices, while the second uses an alphanumeric index and a spatial index.

Plan P2A$_I$ Search picnic tuples and scenic view tuples using the alphanumeric index on `class`. For each picnic tuple, check all scenic view tuples to determine which ones are within the specified distance.

```
get all tuples of logical_images corresponding to "picnic"
    (use index li_cl)
for each such tuple t1
  get all tuples of logical_images corresponding to "scenic view"
    (use index li_cl)
  for each such tuple t2
    if distance between t1 and t2 $\leq$ 5 miles
      and they are in the same image then
      display corresponding physical image
```

Plan P2B$_I$ Search for "picnic" tuples using an alphanumeric index on `class` and search for "scenic view" tuples using a spatial index on `loc`.

```
get all tuples of logical_images corresponding to "picnic"
    (use index li_cl)
for each such tuple t
  get all points within 5 miles of t.loc
    (using the incremental nearest neighbor operation)
  for each one of these points p
    if p is a ''scenic view'' and in same image then
      display the corresponding physical image
```

Plan P2A$_P$ Search both the picnic and scenic view partitions sequentially.

```
for each tuple t1 of the "picnic" partition
  for each tuple t2 of the "scenic view" partition
    if distance between t1.loc and t2.loc $\leq$ 5 miles
        and they are in the same image then
      display the corresponding physical image
```

Plan P2B$_P$ Search the picnic partition sequentially, and search the scenic view partition using the spatial index on `loc`.

```
for each tuple t1 of the "picnic" partition
  get all points within 5 miles of t1.loc
    in the "scenic view" partition
  for each one of these points p
    if p is in the same image as t1 then
      display the corresponding physical image
```

Cost Analysis

In order to estimate the costs of each plan, we must make assumptions about the data distribution and the costs of the various operations. Table 1.1 contains a tabulation of the costs of basic operations used to process queries. The cost of many of these operations is a function of the relation on which they operate. $c_{x(y)}$ is the cost of performing operation x on relation or index y. `li` stands for `logical_images`. The

Name	*Meaning*
c_r	accessing a tuple by tid (random order)
c_{sq}	accessing a tuple in sequential order
c_{sqf}	accessing the first tuple of a relation
c_{af}	"find first" operation on an alphanumeric index
c_{an}	"find next" operation on an alphanumeric index
c_{lsf}	"find nearest neighbor" operation on a location space index
c_{lsn}	"find next nearest neighbor" operation on a location space index
c_{fsf}	"find nearest neighbor" operation on a feature space index
c_{fsn}	"find next nearest neighbor" operation on a feature space index
c_{sc}	string comparison
c_{lsd}	distance computation in location space
c_{fsd}	weighted distance computation in feature space

Table 1.1 Costs of basic operations used in query processing.

cost of accessing the physical_images relation to retrieve the result image and the cost of the "display" operation are not included as it is always the same regardless of the selected execution plan. Let N_{pic} and N_{sv} be the number of tuples from class "picnic" and "scenic view", respectively. Let B_{pic} and B_{sv} be the number of disk blocks containing tuples from class "picnic" and "scenic view", respectively.

Equations 1.1, and 1.2 estimate the cost of responding to query 1 using the integrated and partitioned organizations, respectively.

$$C_{1_I} = c_{af(li\_cl)} + N_{sv} \times (c_{r(li)} + c_{an(li\_cl)}) \tag{1.1}$$
$$C_{1_P} = N_{sv} \times c_{sq(sv\_part)} \tag{1.2}$$

One difference between C_{1_I} and C_{1_P} is that in the integrated organization, there is an "alphanumeric find" operation on index `li_cl` that is not necessary in the partitioned organization. It is required in order to find the first scenic view tuple in this index. In addition, one more random access is required for each scenic view tuple in order to get the `img_id` from the `logical_images` relation. The other difference is that there are N_{sv} alphanumeric next operations in the integrated organization compared with N_{sv} sequential access operations in the partitioned organizations. The reason for this is that in the partitioned organization, the relation is scanned directly, whereas in the integrated organization, the index is scanned.

$$\begin{aligned} C_{2A_I} &= c_{af(li\_cl)} + N_{pic} \times (c_{r(li)} + c_{an(li\_cl)}) + \\ &\quad B_{pic} \times [c_{af(li\_cl)} + N_{sv} \times (c_{r(li)} + c_{an(li\_cl)})] + \\ &\quad N_{pic} \times N_{sv} \times c_{lsd} \end{aligned} \tag{1.3}$$
$$\begin{aligned} C_{2A_P} &= N_{pic} \times c_{sq(pi\_part)} + \\ &\quad B_{pic} \times [c_{sqf(sv\_part)} + N_{sv} \times c_{sq(sv\_part)}] + \\ &\quad N_{pic} \times N_{sv} \times c_{lsd} \end{aligned} \tag{1.4}$$

Equations 1.3 and 1.4 estimate the cost of responding to query 2 with plan A using the integrated and partitioned organizations, respectively. In both equations, the first line is the cost of reading all pic tuples, the second line is the cost of reading all sv tuples for each block, and the last line is the cost of checking the distance between each (pic,sv) pair. N_{InC_2} denotes the average number of tuples in the circular range specified in query 2 (C_2). $N_{sv\_InC_2}$ denotes the average number of scenic view tuples in C_2. Assuming a uniform distribution of symbols in space (i.e., there is an equal number of symbols in any given area), then $N_{InC_2} = \frac{area(C_2)}{A} \times N$, where N is the total number of tuples in the `logical_images` relation, A is the area covered by these tuples, and C_2 is the circular range specified in query 2. Assuming a uniform distribution of classifications among the symbols (i.e., there is an equal number of symbols from each classification in any group of symbols), then $N_{sv\_InC_2} = \frac{area(C_2)}{A} \times \frac{N}{CL}$, where *CL* is the number of different classifications in the database.

If these assumptions about the distribution of the classifications among symbols do not hold, then other methods are required to estimate the number of scenic view tuples in a given area. The portion of all tuples that belong to each classification can be recorded when populating the database by checking the `class` attribute and tallying the number for each classification. This data can then be used to estimate the distribution of the classifications among the symbols. Assuming that the distribution of classifications among any group of symbols is equal to the the total database distribution (i.e., the portion of tuples from each classification among any given group of symbols is equal to the portion of tuples from each classification in the entire database), then $N_{sv\_InC_2} = \frac{area(C_2)}{A} \times sv_p \times N$, where sv_p is the portion of the database tuples that belong to the "scenic view" class.

Plan P2A$_C$ performs a spatial join operation on the results of two selection operations on relation `logical_images`. The first select operation extracts all tuples of the relation that are of class "picnic", while the second select operation extracts all tuples of the relation that are of class "scenic view". The results of these two select operations are then joined according to a predicate based on the `loc` attribute. In our implementation of plan P2A$_C$, we perform the select and join operations simultaneously using a block nested loop join algorithm as follows. One of the classes is designated as the *inner class*, and the other is designated as the *outer class*. One block of tuples belonging into the outer class are read into a memory-resident buffer (using the index on attribute `class`). All tuples of the inner class are then read (one block at a time using the index on attribute `class`) and spatially joined with all tuples of the outer class that are in memory (by computing the predicate on the spatial attribute). This process is repeated with the next block of tuples of the outer class, until all tuples of the outer class have been read.

The main difference between C_{2A_I} and C_{2A_P} is that in the integrated organization the index is scanned sequentially, whereas in the partitioned organization the relation corresponding to the scenic view partition is scanned sequentially (as in the case of query 1). As a result, once again, there are considerably more "random access" operations in the integrated organization than in the partitioned organization.

Equations 1.5 and 1.6 estimate the cost of responding to query 2 with plan B using the integrated and partitioned organizations, respectively. Again, as in equations 1.3 and 1.4, the first line is the cost of reading all pic tuples, but the second line is the cost of finding sv tuples in the range (using index li_loc) for each pic.

$$\begin{aligned} C_{2B_I} &= c_{af(li\_cl)} + N_{pic} \times [c_{r(li)} + c_{an(li\_cl)}] + \\ &\quad N_{pic} \times [c_{lsf(li\_loc)} + N_{InC_2} \times (c_{r(li)} + c_{sc} + c_{lsn(li\_loc)})] \end{aligned} \quad (1.5)$$

$$\begin{aligned} C_{2B_P} &= N_{pic} \times c_{sq(pi\_part)} + \\ &\quad N_{pic} \times c_{lsf(sv\_loc)} + N_{sv\_InC_2} \times (c_{r(sv\_part)} + c_{lsn(sv\_loc)}) \end{aligned} \quad (1.6)$$

The main difference between C_{2B_I} and C_{2B_P} is in the number of location-space “find next” operations and the number of random access operations. In the integrated organization, all tuples `t` of any class in circle C_2 are retrieved from the spatial index. The `class` of `t` is then retrieved from the `logical_images` relation to see if it corresponds to a “scenic view”. This requires a random access operation for each tuple in C_2. On the other hand, in the partitioned organization, only tuples of type “scenic view” are retrieved by the spatial index. Thus, there is no need for an additional random access to check the class of the tuple. In addition, since only a subset of the tuples in circle C_2 are “scenic view” tuples, the number of items retrieved by the spatial query in the integrated organization (i.e., N_{InC_2}) is larger than the number of items retrieved by the spatial query in the partitioned organization (i.e., $N_{sv\_InC_2}$).

Another significant difference between C_{2B_I} and C_{2B_P} is that the spatial index on which the search is performed is smaller in the partitioned organization since it only contains “scenic view” tuples (i.e., $|\texttt{sv\_loc}| < |\texttt{li\_loc}|$). As a result, $c_{lsf(sv\_loc)}$ and $c_{lsn(sv\_loc)}$ are less than $c_{lsf(li\_loc)}$ and $c_{lsn(li\_loc)}$, respectively. Therefore, the difference between the total cost of plan P2B in the partitioned organization and the total cost of plan P2B in the integrated organization is greater than in the case of plan P2A. The plan for the partitioned organization can be further improved by implementing a more sophisticated form of the *spatial join* operation between the two relations `scenic_part` and `pi_part` which correspond to “scenic view” and “picnic”, respectively. The overall idea is that the join can be computed more efficiently by traversing both indices in parallel in such a way as to avoid comparing tuples which cannot satisfy the join condition. This operation has not been implemented in SAND yet. Once it is added, plan $P2B_P$ will be revised accordingly.

It is interesting to compare the costs of answering query 2 for one particular organization using plans P2A and P2B. For the integrated organization, we compare equations 1.3 and 1.5. In plan P2A$_I$, both relations are scanned sequentially via the alphanumeric index `li_cl`. For each picnic tuple, each scenic view tuple is checked to determine whether or not it is within the specified range. Thus, the total number of distance computations is $N_{pic} \times N_{sv}$. In addition, the same number of random access operations are also required in order to get the locations from the `logical_images` relations. In plan P2B$_I$, the spatial index is used and thus only tuples that are within the specified range need to be examined. The cost of this is the overhead involved in using the spatial index. In this case, this cost is N_{pic} location-space “find first” operations, and $N_{pic} \times N_{InC_2}$ location-space “find next” operations. These spatial operations involve distance computations as part of the incremental nearest neighbor operation. However, there is no need for any distance computations as part of the plan itself. Whether plan P2A$_I$ or plan P2B$_I$ is better depends on the size of the data set, the portion of these tuples that belong to each classification (termed the *contextual selectivity*), and on the portion of all tuples that fall in the range specified by the spatial component (termed the *spatial selectivity*). Assuming a high spatial selectivity (i.e., that the number of tuples in the spatial range is much smaller than the total number of tuples in the data set), plan P2B$_I$ should prove to be much more efficient than plan P2A$_I$. However, if the spatial selectivity is low, then plan P2A$_I$ may prove to be better. Similar observations can be made about the partitioned organization by comparing equations 1.4 and 1.6. Once a more efficient spatial join operator is implemented, as mentioned above, the difference will be even greater.

CONCLUDING REMARKS

Two different data organizations (integrated and partitioned) for storing logical images in relational tables were proposed. They differ in the way that the logical images are stored. Sample queries and execution plans to answer these queries were described for both organizations. Analytical cost analyses of these execution plans were given that indicated that the partitioned data organization is more efficient for queries that consist of both contextual and spatial specifications. On the other hand, the integrated organization is better for purely spatial specifications. Both organizations gave similar results for queries that consist of purely contextual specification.

Our definition of the class of images that we can handle is rather strict. Some of these restrictions can be relaxed. In particular, the requirement that there exists a function f which when given a symbol s and a class C returns a value between 0 and 1 indicating the certainty that s belongs to C can be omitted. In this case, we can store the feature vectors in the database rather than the classifications. For a comparison of using these two approaches, see [16]. Of course, more elaborate indexing methods are then required to respond to queries such as those presented in this paper.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grants CDA-950-3994 and IRI-97-12715. We are grateful to Karttakeskus, Map Center, Helsinki, Finland for providing us the map data.

References

[1] W. G. Aref and H. Samet. Optimization strategies for spatial query processing. In *Proc. of the 17th Intl. Conf. on Very Large Data Bases*, pp. 81–90, Barcelona, Sept. 1991.

[2] S. K. Chang, E. Jungert, and Y. Li. The design of pictorial databases based upon the theory of symbolic projections. In *Design and Implementation of Large Spatial Databases — 1st Symp., SSD'89*, pp. 303–323, Santa Barbara, CA, July 1989. (Also Springer-Verlag Lecture Notes in Computer Science 409).

[3] C. Esperança and H. Samet. Spatial database programming using SAND. In

Proc. of the 7th Intl. Symp. on Spatial Data Handling, vol. 2, pp. A29–A42, Delft, The Netherlands, Aug. 1996.

[4] C. Faloutsos. Access methods for text. *ACM Comp. Surveys*, 17(1):49–74, Mar. 1985.

[5] V. Gudivada and V. Raghavan. Design and evaluation of algorithms for image retrieval by spatial similarity. *ACM Trans. Info. Syst.*, 13(2):115–144, Apr. 1995.

[6] A. Gupta, T. Weymouth, and R. Jain. Semantic queries with pictures: the VIMSYS model. In *Proc. of the 17th Intl. Conf. on Very Large Databases*, pp. 69–79, Barcelona, Spain, Sept. 1991.

[7] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Advances in Spatial Databases — 4th Intl. Symp., SSD'95*, pp. 83–95, Portland, ME, Aug. 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951).

[8] R. Jain. NSF workshop on visual information management systems. *SIGMOD RECORD*, 22(3):57–75, Sept. 1993.

[9] R. Kasturi, R. Raman, and C. Chennubhotla. Document image analysis an overview of techniques for graphics recognition. In *Proc. of the IAPR Workshop on Syntactic and Structural Pat. Rec.*, pp. 192–230, Murray Hill, NJ, June 1990.

[10] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, Aug. 1986. (Also *Proc. of SIGGRAPH'86*, Dallas, Aug. 1986).

[11] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, and P. Yanker. The QBIC project: Querying images by content using color, texture, and shape. In *Proc. of the SPIE, Storage and Retrieval of Image and Video Databases*, vol. 1908, pp. 173–187, San Jose, CA, Feb. 1993.

[12] V. Oria, B. Xu, and M. T. Tamer. VisualMOQL: A visual query language for image databases. In *Proc. of the IFIP 2.6 4th Working Conf. on Visual Database Systems (VDB-4)*, pp. 186–191, L'Aquila, Italy, May 1998.

[13] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. In *Proc. of the SPIE, Storage and Retrieval of Image and Video Databases II*, vol. 2185, pp. 34–47, San Jose, CA, Feb. 1994.

[14] D. Rotem. Spatial join indices. In *Proc. of the 7th Intl. Conf. on Data Eng.*, pp. 500–509, Kobe, Japan, April 1991. IEEE Computer Society Press.

[15] H. Samet and A. Soffer. MAGELLAN: Map acquisition of geographic labels by legend analysis. *Intl. Journal of Document Analysis and Recognition*, 1(2):89–101, June 1998.

[16] A. Soffer and H. Samet. Two approaches for integrating symbolic images into a multimedia database system: a comparative study. *VLDB Journal*, to appear.

[17] M. Swain. Interactive indexing into image databases. In *Proc. of the SPIE, Storage and Retrieval for Image and Video Databases*, vol. 1908, pp. 95–103, San Jose, CA, Feb. 1993.

[18] Virage. Virage web site. http://www.virage.com.