PREPRINT – Please don't distribute.
To appear in Vittorio Castelli and Lawrence Bergman eds.,
"Image Databases, Search and Retrieval of Digital Imagery",
2001, John Wiley and Sons.

Database Support for Multimedia Applications

Michael Ortega-Binderberger, Kaushik Chakrabarti
University of Illinois at Urbana-Champaign

Sharad Mehrotra
University of California at Irvine

August 3, 2001

1 Introduction

Advances in high performance computing, communication, and storage technologies, as well as emerging large-scale multimedia applications, have made the design and development of multimedia information systems one of the most challenging and important directions of research and development within computer science. The payoffs of a multimedia infrastructure are tremendous-it enables many multi-billion dollar-a-year application areas. Examples are medical information systems, electronic commerce, digital libraries, (like multimedia data repositories for training, education, broadcast and entertainment,) special purpose databases, (such as face/fingerprint databases for security,) and geographical information systems storing satellite images, maps, etc.

An integral component of the multimedia infrastructure is a *multimedia database management system*. Such a system supports mechanisms to extract and represent the content of multimedia objects, provides efficient storage of the content in the database, supports content-based queries over multimedia objects, and provides a seamless integration of the multimedia objects with the traditional information stored in existing databases. A multimedia database system consists of multiple components, which provide the following functionalities:

- **Multimedia Object Representation:** techniques/models to succinctly represent both structure and content of multimedia objects in databases.
- **Content Extraction:** mechanisms to automatically/semi-automatically extract meaningful features that capture the content of multimedia objects, and that can be indexed to support retrieval.
- **Multimedia Information Retrieval**: techniques to match and retrieve multimedia objects based on the similarity of their representation (i.e., similarity-based retrieval).
- **Multimedia Database Management:** extensions to data management technologies of indexing and query processing to effectively support efficient content-based retrieval in database management systems.

Many of the above issues have been extensively addressed in other chapters of this book. Our focus in this chapter is on how content-based retrieval of multimedia objects can be integrated into database management systems as a primary access mechanism. In this context, we first explore the support provided by existing object-oriented and object-relational systems for building multimedia applications. We then identify limitations of existing systems in supporting content-based retrieval and summarize approaches proposed in the literature to address these limitations. We believe that this research will culminate in improved data management products that support multimedia objects as "first-class" objects, capable of being efficiently stored and retrieved based on their internal content.

The rest of the chapter is organized as follows. In Section 2, we describe a simple model for content-based retrieval of multimedia objects, which is widely implemented and commonly supported by commercial vendors. We use this model throughout the chapter to explain the issues that arise in integrating content-based retrieval into database management systems (DBMSs). In Section 3 we explore how the evolution of relational databases into object-oriented and object-relational systems, which support complex data types and user-defined functions, facilitates building multimedia applications [104]. We apply the analysis framework of Section 3 to the Oracle, the Informix, and the IBM DB2 database systems in Section 4. The chapter then identifies limitations of existing state-of-the-art data management systems from the perspective of supporting multimedia applications. Finally, Section 5 outlines a set of research issues and approaches that we believe are crucial for the development of database technology providing seamless support for complex multimedia information.

2 A Model for Content-Based Retrieval

Traditionally, content-based retrieval from multimedia databases was supported by describing multimedia objects with textual annotations [90, 101, 37, 58]. Textual information retrieval techniques [92, 61, 65, 35] were then used to search for multimedia information indirectly using the annotations. Such a *text-based approach* suffers from numerous limitations, including the impossibility of scaling it to large data sets (due to the high degree of manual effort required to produce the annotations), the difficulty of expressing visual content (e.g., texture/patterns or shape in an image) using textual annotations, and the subjectivity of manually generated annotations.

To overcome several of these limitations, a *visual feature-based approach* has emerged as a promising alternative, as is evidenced by several prototype [86, 52, 100] and commercial systems [38, 33, 30, 6, 55]. In a visual feature-based approach, a multimedia object is represented using visual properties; for example, a digital photograph may be represented using color, texture, shape, and textual features. Typically, a user formulates a query by providing examples, and the system returns the "most similar" objects in the database. The retrieval consists of ranking the similarity between the feature-space representations of the query and of the images in the database. The query process can therefore be described by defining the models for objects, queries, and retrieval.

2.1 Object Model

A multimedia object is represented as a collection of extracted features. Each feature may have multiple representations, capturing it from different perspectives. For instance, the color histogram [105] descriptor represents the color distribution in an image using value counts, while the color moments [51] descriptor represents the color distribution in an image using statistical parameters (e.g., mean, variance, and skewness). Associated with each representation is a similarity function that determines the similarity between two descriptor values. Different representations capture the same feature from different perspectives. The simultaneous use of different representations often improves retrieval effectiveness [52], but it also increases the dimensionality of the

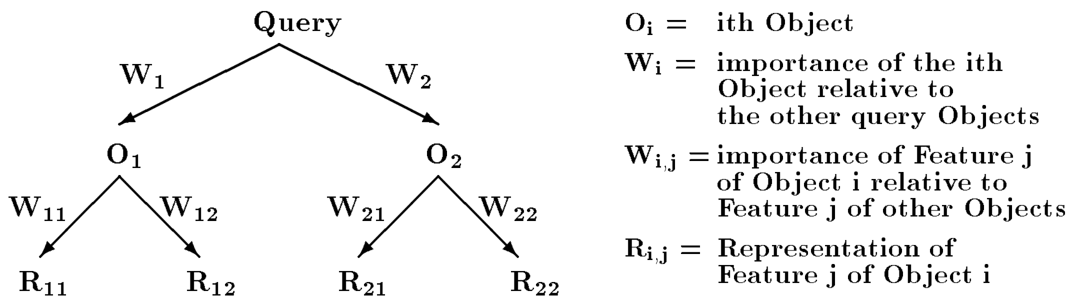
Query
W1
W2
O1
O2
W11
W12
W21
W22
R11
R12
R21
R22
Oi = ith Object
Wi = importance of the ith Object relative to the other query Objects
Wi,j = importance of Feature j of Object i relative to Feature j of other Objects
Ri,j = Representation of Feature j of Object i

Figure 1: Query Model

search space (which reduces retrieval efficiency), and has the potential for introducing redundancy (which can negatively affect effectiveness.)

Each feature space (e.g., a color histogram space) can be viewed as a multidimensional space, in which a feature vector representing an object corresponds to a point. A metric on the feature space can be used to define the dissimilarity between the corresponding feature vectors. Distance values are then converted to similarity values. Two popular conversion formulae are $s = 1 - d$[1] and $s = exp(-\frac{d^2}{2})$, where s and d respectively denote similarity and distance. With the first formula, if d is measured using the *Euclidean distance function*, s becomes the *cosine similarity* between the vectors, while if d is measured using the *Manhattan distance function*, s becomes the *histogram intersection similarity* between them. While cosine similarity is widely used in keyword-based document retrieval, histogram-intersection similarity is common for color histograms. A number of image features and feature matching functions are further described in Chapters 8 to 19.

2.2 Query Model

The query model specifies how a query is constructed and structured. Much like multimedia objects, a query is also represented as a collection of features. One difference is that a user may simultaneosly use multiple example-objects, in which case the query can be represented in either of the following two ways [89]:

- **Feature-based representation:** The query is represented as a collection of features. Each feature contains a collection of feature representations with multiple values. The values correspond to the feature descriptors of the objects.

- **Object-based representation:** A query is represented as a collection of objects and each object consists of a collection of feature descriptors.

In either case, each component of a query is associated with a weight indicating its relative importance.

Figure 1 shows a structure of a query tree in an object-based model. In the figure, the query structure consists of multiple objects O_i and each object is represented as a collection of multiple feature values R_{ij}.

[1]The conversion formula assumes that the space is normalized to guarantee that the maximum distance between points is equal to 1.

2.3 Retrieval Model

The retrieval model determines the similarity between a query tree and objects in the database. The leaf level of the tree corresponds to feature representations. A similarity function specific to a given representation is used to evaluate the similarity between a leaf node (R_{ij}) and the corresponding feature representation of the objects in the database. Assume, for example, that the leaf nodes of a query tree correspond to two different color representations — color histogram and color moments. While histogram intersection [105] may be used to evaluate the similarity between the color histogram of an object and that of the query, the weighted Euclidean distance metric may be used to compute the similarity between the color moments descriptor of an object and that of the query. The matching (or retrieval) process at the feature representation level produces one ranked list of results for each leaf of the query tree. These ranked lists are combined using another function to generate a ranked list describing the match results at the parent node. Different functions may be used to merge ranked lists at different nodes of the query tree, resulting in different retrieval models. A common technique used is the *weighted summation model.* Let a node N_i in the query tree have children N_{i1} to N_{in}. The similarity of an object O in the database with node N_i (represented as $similarity_i$) is computed as:

$$\begin{aligned} similarity_i &= \sum_{j=1}^{n} w_{ij}\ similarity_{ij} \qquad \text{where} \\ \sum_{j=1}^{n} w_{ij} &= 1 \end{aligned} \tag{1}$$

and $similarity_{ij}$ is the measure of similarity of the object with the jth child of node N_i.

Many other retrieval models to generate overall similarity between an object and a query have been explored in the literature. For example, in [82], a Boolean model suitably extended with fuzzy and probabilistic interpretations is used to combine ranked lists. A Boolean operator — AND ($\wedge$), OR ($\vee$), NOT ($\neg$) — is associated with each node of the query tree, and the similarity is interpreted as a fuzzy value or a probability and combined with suitable merge functions. Desirable properties of such merge functions are studied by Fagin and Wimmers in [32].

2.4 Extensions

In the previous section, we have described a simple model for content-based retrieval that will serve as the base reference in the remainder of the chapter. Many extensions are possible and have been proposed in the literature. For example, we have implicitly assumed that the user provides appropriate weights for nodes at each level of the query tree (reflecting the importance of a given feature/node to the user's information need [92]). In practice, however, it is difficult for a user to specify the precise weights. An approach followed in some research prototypes (i.e., MARS [52], MindReader [57]) is to learn these weights automatically using the process of *relevance feedback* [89, 88, 91]. Relevance feedback is used to modify the query representation by altering the weights and structure of the query tree to better reflect the user's subjective information need.

Another limitation of our reference model is that it focuses on representation and content-based retrieval of images — it has limited ability to represent structural, spatial or temporal properties of general multimedia objects (i.e., multiple synchronized audio and video streams) and to model retrieval based on these properties. Even in the context of image retrieval, the model described needs to be appropriately extended to support a more structured retrieval based on local/region-based properties. Retrieval based on local region-specific properties and the spatial relationships between the regions has been studied in many prototypes including [71, 17, 83, 99, 67].

3 Overview of Current Database Technology

In this section, we explore how multimedia applications requiring content-based retrieval can be built using existing commercial data management systems. Traditionally, relational database technology has been geared towards business applications where data is largely in tabular form with simple atomic attributes. Relational systems usually support only a handful of data types — a numeric type with its usual variations in precision,[2] a text type with some variations in the assumptions about the storage space available,[3] some temporal data types such as date and time with some variations[4]. Providing support for multimedia objects in relational database systems poses many challenges. First, in contrast to the limited storage requirements of traditional data types, multimedia data such as images, video, and audio are quite voluminous — a single record may span several pages. One alternative is to store the multimedia data in files *outside* of the DBMS control with only *pointers* or references to the multimedia object stored in the DBMS. This approach has numerous limitations since it makes the task of optimizing access to data difficult, and furthermore prevents DBMS access control over multimedia types. An alternative solution is to store the multimedia data in databases as *binary large objects* (BLOBs), which are supported by almost all commercial systems. BLOB is a data type used for data that does not fit into one of the standard categories, because of its large size or its widely variable length, or because the only needed operation is storage, rather than interpretation, analysis or manipulation.

While modern databases provide effective mechanisms to store very large multimedia objects in a BLOB, BLOBs are uninterpreted sequences of bytes, which cannot represent the rich internal structure of multimedia data. Such a structure can be represented in a DBMS using the support for user-defined abstract data types (ADTs) offered by modern object-oriented and object-relational databases. Such systems also provide support for user-defined functions (UDFs) or methods, which can be used to implement similarity retrieval for multimedia types. Similarity models, implemented as UDFs, can be called from within SQL allowing content-based retrieval to be seamlessly integrated into the database query language. In the remainder of this section, we discuss the support for ADTs, UDF, and BLOBs in modern databases that provides the core technology for building multimedia database applications.

3.1 User-Defined Abstract Data Types

The basic relational model requires tables to be in the first normal form [27] where every attribute is atomic. This poses serious limitations in supporting applications that deal with objects/data types with rich internal structure. The only recourse is to translate between the complex structure of the applications and the relational model every time an object is read or written. This results in extensive overhead making the relational approach unsuitable for advanced applications that require support for complex data types.

[2]Typically, numeric data can be of integral type, fractional data such as floating point in various precisions, and specialized money types such as packed decimal that retained high precision for detailed money transactions.

[3]Notably, the *char* data type specifies a maximum length of a character string and this space is always reserved. *Varchar* data in contrast occupies only the needed space for the stored character string and also has a maximum length.

[4]Variations of temporal data types include *time*, *date*, *datetime* sometimes with a precision specification such as year down to hours, *timestamp* used to mark a specific time for an event, and *interval* to indicate the length of time.

These limitations of relational systems have resulted in much research and commercial development to extend the database functionality with rich user-defined data types in order to accommodate the needs of advanced applications. Research in extending the relational database technology has proceeded along two parallel directions.

The first approach, referred to as the object-oriented database (OODBMS) approach, attempts to enrich object-oriented languages, such as C++ and Smalltalk, with the desirable features of databases, such as concurrency control, recovery, and security, while retaining support for the rich data types and semantics of object-oriented languages. Examples of systems that have followed this approach include research prototypes such as [16] and a number of commercial products [7, 66].

The object-relational database (ORDBMS) systems, on the other hand, approach the problem of adding additional data types by extending the existing relational model with the full-blown type hierarchy of object-oriented languages. The key observation was that the concept of domain of an attribute need not be restricted to simple data types. Given its foundation in the relational model, the ORDBMS approach can be considered a less radical evolution than the OODBMS approach. The ORDBMS approach produced such research prototypes as Postgres [103], and Starburst [46] and commercial products such as Illustra [104]. The ORDBMS technology has now been embraced by all major vendors including Informix [53], IBM DB2 [22], Oracle [74], Sybase [107], and UniSQL [60] among others. The ORDBMS model has been incorporated in the SQL-3 standards.

While OODBMSs provide the full power of an object-oriented language, they have lost ground to ORDBMSs. Interested readers are referred to [104] for good insight from both a technical and commercial perspective into reasons for this development. In the remainder of this chapter, we will concentrate on the ORDBMS approach.

The object-relational model retains relational model concepts of tables and columns in tables. Besides the basic types, it provides for additional user-defined abstract data types (ADTs), as well as collections of basic and user-defined types. The functions that operate on these ADTs, known as *User-Defined Functions* (UDFs), are written by the user and are equivalent to *methods* in the object-oriented context. In the object-relational model, the fields of a table may correspond to basic DBMS data types, to other ADTs, or can even just contain storage space whose interpretation is entirely left to the user-defined methods for the type [53]. The following example illustrates how a user may create an ADT and include it in a table definition:

```
create type ImageInfoType ( date varchar(12) ,
                            location_latitude real ,
                            location_longitude real )

create table SurveyPhotos ( photo_id integer primary key not null,
                            photographer varchar(50) not null,
                            photo_location ImageInfoType not null,
                            photo blob not null)
```

The type *ImageInfoType* defines a structure to store the location at which a photograph was taken together with the date stored as a string. This can be useful for nature survey applications where a biologist may wish to attach a geographic location and a date to a photograph. This abstract data type is then used to create a table with an id for the photograph, the photographer's name, the photograph itself (stored as a BLOB), and the location and date when it was taken.

ORDBMSs extend the basic SQL language to allow user-defined functions (once they are compiled and registered with the DBMS) to be called directly from within SQL queries, thereby providing a natural mechanism to develop domain-specific extensions to databases. The following example shows a sample query that calls a user-defined function on the type declared above:

```
select photographer, convert_to_grayscale(photo)
       from SurveyPhotos
       where within_distance(photo_location,'1', '30.45, -127.0')
```

This query returns the photographer and a gray-scale version of the image stored in the table. The *within_distance* UDF is a predicate that returns true if the place where the image was shot is within 1 mile of the given location. This UDF ignores the date the picture was taken, demonstrating how predicates are free to implement any semantically significant properties of an application. Note that the UDF *convert_to_grayscale* to convert the image to gray-scale is not a predicate since it is applied to an attribute in the *select* clause and returns a gray-scale image.

ADTs also provide for type inheritance and, as a consequence, polymorphism. This introduces some problems in the storage of ADTs, as existing storage mangers assume that all rows in a table share the same structure. Several strategies have been developed to cope with this problem [39], including dynamic interpretation, and using distinct physical tables for each possible type of a larger, logical table. Section 5.1 contains more details on this topic.

3.2 Binary Large Objects

As mentioned previously, binary large objects (BLOBs) are used for data that does not fit into any of the conventional data types supported by a DBMS. BLOBs are used as a data type for objects that are either large, have wildly varying size, cannot be represented by a traditional data type, or whose data might be corrupted by character table translation.[5] Two main characteristics set BLOBs apart from other data types: they are stored separately from the record [23] and their data type is just a string of bytes.

BLOBs are stored separately due to their size: if placed in-line with the record, they could span multiple pages and hence introduce loss of clustering in the table storage. Furthermore, applications may frequently only choose to access other attributes and not BLOBs — or access BLOBs selectively based on other attributes. Indeed, BLOBs have a different access pattern than other attributes. As observed in [59], it is unreasonable to assume that applications will read and/or update all the bytes belonging to a BLOB at once. It is more reasonable to assume that only portions or substrings (byte or bit) will be read or updated during individual operations. To cope with such an access pattern, many DBMSs distinguish between two types of BLOBs:

- *regular BLOBs*, in which the application receives the whole data in a host variable all at once, and
- *smart BLOBs*, in which the application receives a handle, and uses it to read from the BLOB using the well-known file system interfaces *open*, *close*, *read*, *write*, and *seek*. This allows fine-grained access to small parts of the BLOB.

Besides the above two mechanisms to deliver BLOBs from the database to applications (that is, either via whole chunks or via a file interface), a third option of a streaming interface is also possible. Such an interface is important for guaranteeing timely delivery of continuous media objects, such

[5]Most DBMSs support data types that could be used to store objects of miscellaneous types. For example, a small image icon can be represented using a *varchar* type. The icon would be stored in-line with the record instead of separately (as would be the case if the image icon is stored as a BLOB). Even though there may be performance benefits from storing the icon in-line (say it is very frequently accessed), it may still not be desirable to store it as a *varchar* since the icon may get corrupted in transmission and interpretation across different hardware (due to the differences in character set representation across different machines). Such data types, sensitive to character translation, should be stored as BLOBs.

as audio or video. Currently, to the best of our knowledge, no DBMS offers a streaming interface to BLOBs. Continuous media objects are stored outside the DBMSs in specialized storage servers [13] and accessed from applications directly and not through a database interface. This may, however, change with the increasing importance of continuous media data in enterprise computing.

BLOBs present an additional challenge during query processing. Unless a BLOB is part of a query predicate, it is best to avoid the inclusion of the corresponding column during query processing, since it saves an extra file access during processing, and, more importantly, since BLOBs, due to their size, tend to thrash the database buffers used for query processing. For this reason, BLOB handles are often used and, when the user requests the BLOB content, separate database buffers are used to complete this transfer.

For access control purposes, BLOBs are treated as a single atomic field in a record. Large BLOBs could, in principle, be shared by multiple users, but the most fine grained locking unit in current databases is a tuple (or row) lock, which simultaneously locks all the fields inside the tuple, including the BLOBs. Some of the SQL extensions needed to support parallel operations from applications into database systems are discussed in [41].

3.3 Support for Extensible Indexing

While user-defined ADTs and UDFs provide adequate modeling power to implement advanced applications with complex data types, the existing access methods that support the traditional relational model (i.e., B-tree and hashing) may not provide efficient retrieval of these data types. Consider, for example, a data type corresponding to the geographical location of an object. A spatial data structure such as an R-tree [45] or a grid file [72] might provide a much more efficient retrieval of objects based on spatial location than a collection of B-trees each indexing a separate spatial dimensions. Access methods that exploit the semantics of the data type may reduce the cost of retrieval. As discussed in Chapters 14 and 15, this is certainly true for multimedia types such as images where features (i.e., color, texture, and shape) used to model image content correspond to high-dimensional feature spaces. Retrieval of multimedia objects based on similarity in these feature spaces cannot be adequately supported using B-trees or, for that matter, common multi-dimensional data structures such as R-tree and region quad-tree that are currently supported by certain commercial DBMSs. Specialized access methods (see Chapter 14) need to be incorporated into the DBMS to support efficient content-based retrieval of multimedia objects.

Commercial ORDBMS vendors support extensible access methods [11, 102] since it is not feasible to provide native support for all possible type-specific indexing mechanisms. These type-specific access methods can then be used by the query processor to access data (that is, implement type specific UDFs) efficiently. While these systems support extensibility at the level of access methods, the interface exported for this purpose is at a fairly low level and requires that access method implementors write their own code to pack records into pages, maintain links between pages, handle physical consistency as well as concurrency control for the access method etc. This makes access method integration a daunting task. Other (cleaner) approaches to adding new type-specific access methods are currently a topic of active research [47] and will be discussed in Section 5.2.3.

3.4 Integrating External Data Sources

Many data sources are external to database systems, therefore it is important to extend querying capabilities to such data. This can be accomplished by providing a relational interface to external data and making it look like tables, or by storing external data in the database while maintaining an external interface for traditional applications to access the data. These two approaches are discussed next in more detail.

External data can be made to appear as an internal database table by registering user-defined functions that access resources external to the database server, even including remote services such as search engines, remote servers, etc. For example, Informix has extended its Universal Server to offer the capability of "Virtual Tables" (VTI), in which the user defines a set of functions designed to access an external entity and make it appear to be a normal relational table suitable for searching and updating. Similarly, DB2 uses table functions and special SQL *TABLE* operators to simulate the existence of an internal table. The primary aim of the table functions is to access external search engines to assist DB2 in computing the answers for a query. A detailed discussion of their support is found in [28].

Another approach to integrate external data is based on the realization that much unstructured data (up to 90%) resides outside of DBMSs. This led several vendors to develop a way to extend their database offerings to incorporate such external data into the database while maintaining its current functional characteristics intact. IBM developed an extension to their DB2 database named *Datalinks*, in which a DBMS table can contain a column, which is an "external file." This file is accessible by the table it logically resides in, and through the traditional file system interface. Users have the illusion of interacting with a file system with traditional file system commands while the data is stored under DBMS control. In this way, traditional applications can still access their data files without restrictions, and enjoy the recovery and protection benefits of the DBMS. This functionality implies protection against data corruption.

Similarly, the Oracle Internet File System [79, 78] addresses the same problem by modifying the file system to store files in database tables as BLOBs. The Oracle Internet File System is of interest here because it allows normal users, including web servers, to access images through file system interfaces, while retaining all DBMS advantages.

These advantages translate into small changes to existing delivery infrastructure such as web servers and text processing programs, while retaining advanced functionality including searching, storage management and scalability.

3.5 Commercial Extensions to DBMSs

We have discussed the evolution of the traditional relational model to modern extensible database technology that supports user-defined abstract data types and functions, and the ability to call such functions from SQL. These extensions provide a powerful mechanism for third-party vendors to develop domain-specific extensions to the basic data management system. Such extensions are called *Datablades* in Informix, *Data Cartridges* in Oracle, and *Extenders* in DB2. Many Datablades are commercially available for the Informix Universal Server — some of which are shipped as standard packages while others can be purchased separately. Example Datablades include the *Geodetic datablade* that supports all the important data types and functions for geospatial applications, and includes an R-tree implementation for indexing spatio-temporal data types. Other Datablades available are the Timeseries Datablade for time-varying numeric data such as stocks, the Web Datablade that provides a tight coupling between the database server and a web server, and a Video Foundation Datablade to handle video files, among others. Similar Cartridges and Extenders are also available for Oracle and DB2 respectively.

Besides commercially available Datablades/Cartridges/Extenders, users can develop their own domain-specific extensions. For this purpose, each DBMS supports an API that a programmer must conform to in developing the extensions. Details of the API offered by Informix can be found in [54]. The API supported by Oracle (referred to as the Oracle Data Cartridge Interface (ODCI)) is discussed in [77].

While each of the different systems (that is, Informix, Oracle, and DB2) support the notion of extensibility, they differ somewhat in the degree of control and protection offered. Informix supports extensibility at a low level with very fine-grained access to the database server. There are a considerable number of hooks into the server to customize many aspects of query processing. For example, for predicates involving user-defined functions over user-defined types[6] the predicate functions have access to the conditions in the where clause itself. This level of access allows for very flexible functionality and speed, at a certain cost in safety — Informix relies on the developers of Datablades to follow their protocol closely and not do any damage. Another feature offered by the Informix Datablade API is allowing UDFs to acquire and maintain memory across multiple invocations. Memory is released by the server based on the duration specified by the data type (i.e., transaction duration, query duration, etc.). Such a feature simplifies the task of implementing certain user-defined functions (i.e., user-level aggregation and grouping operators).

While Informix offers a potentially more powerful model for extensibility, IBM DB2 is the only system that isolates the server from faults in UDFs by allowing the execution of UDFs in their own separate address space [22] in addition to the server address space. With this fine-grained fault containment, errors in UDFs will not bring the database server off-line.

4 Image Retrieval Extensions to Commercial DBMSs

In this section, we discuss the image retrieval extensions available in commercial systems. We specifically explore the image retrieval technologies supported by Informix Universal Server, Oracle, and IBM DB2 products. These products offer a wide variety of desirable features designed to provide integrated image retrieval in databases. We illustrate some of the functionalities offered by discussing how applications requiring image retrieval can be built in these systems. While other vendors support a subset of the desired technologies, none integrate them to the same degree — resulting in a large effort on the part of customers wishing to create multimedia applications.

To demonstrate how image retrieval applications can be built using database extensions for commercial DBMSs, we will use a very simple example of a digital catalog of color pictures. In this application, a collection of pictures, is stored into a table. For each picture, the photographer and date are stored into a table. The basic table schema is:

- **photo_id**: an integer number to identify the item in the catalog
- **photographer**: a character string with the photographer's name
- **date**: the date the picture was taken, for simplicity we will use a character string instead of a date datatype
- **photo**: the photo image and its necessary features for retrieval

The implementation of the *photo* attribute changes depending on the product and is described in the following subsections. In addition to these attributes, any additional attributes, tables and steps necessary to store such a catalog in the database and execute content-based retrieval queries will be illustrated in the subsections below corresponding to the three systems discussed.

4.1 Informix Image Retrieval Datablade

The Informix system includes a complete media asset management suite (called Informix Media360(TM) [56]) to manage digital content in a central repository. The product is designed to

[6]These are special user-defined functions declared as operators.

handle any media type, including images, maps, blueprints, audio, and video, and is extensible to support additional media types. It manages the entire life cycle of media objects, from production, to delivery, to archiving, including access control and rights management. The product is integrated with image, video, and audio catalogers and image, video key-frame, and audio content-based search functionality. This suite includes asset management software and a number of content-specific Datablades to tackle datatype-specific needs. The Excalibur Visual Retrievalware Datablade [55] is one such type-specific Datablade that manages the storage, transcoding and content-based search of images. The image Datablade is also used for video key-frame search. Image retrieval based on color, texture, shape, brightness layout, color structure and image aspect ratio is supported. Color refers to the global color content of the image (i.e., regardless of its location). Texture seeks to distinguish such properties as smoothness or graininess of a surface. Shape seeks to express the shape of objects in an image: for example a balloon is a circular shape. Brightness layout captures the relative energy in an image based on its location in the image, and similarly, color structure seeks to localize the color properties to regions of the image.

For each image in the database, a similarity score is computed to determine the degree to which that image satisfies the query. All feature-to-feature matches are weighted with user-supplied weights and combined into a final score. Only those images with a score above a given similarity threshold are returned to the user and the remaining images are deemed not relevant to the query. The Datablade supports datatypes to store images and their image feature vectors. Feature vectors combine all the feature representations supported into a single attribute for the whole image. Therefore, no sub-image or region searching is possible.

In order to build an image retrieval application using the image datablade in Informix, the following tasks must be performed:

1. Install Informix with the Universal Data Option and the Excalibur Visual Retrievalware Datablade product, then configure the necessary table and index storage space in the server.

2. Create a Database to store all tables and auxiliary data needed for our example. We will call this the *Gallery* database.

   ```
   CREATE DATABASE Gallery;
   ```

3. Create a table with the desired fields, of which two are for the image retrieval. Following our example, this statement creates such a table:

   ```
   CREATE TABLE photo_collection (
                photo_id integer primary key not null,
                photographer varchar(50) not null,
                date varchar(12) not null,
                photo IfdImgDesc not null,
                fv IfdFeatVect)
   ```

   The *photo* field stores the image descriptor and the *fv* field stores the feature vector for the image, which will be used for content-based search.

4. Insert data into the table with all the values except for the *fv* field, which will be filled elsewhere:

   ```
   INSERT INTO photo_collection (photo_id, photographer, date, photo) VALUES
               (3, 'Ansel Adams', '03/06/1995',
                IfdImgDescFromFile('/tmp/03.jpg'))
   ```

Notice that the feature vector attribute was not specified and thus retains a value of NULL. More photo collection entries can be added using this method.

5. At a later time, the features are extracted to populate the *fv* attribute in the table:

```
UPDATE photo_collection
       SET fv = GetFeatureVector(photo)
       WHERE fv IS NULL
```

   This command sets the feature vector attribute for tuples where the features have not yet been extracted, i.e., where the *fv* attribute is NULL. The features are extracted from each *photo* with the *GetFeatureVector* user-defined function that is part of the Datablade. Manually extracting the feature information and updating it in the table is desirable if many images are loaded quickly and feature extraction can be performed at a later time. An alternative to manual feature extraction is to automatically extract the features when each tuple is inserted or updated. To accomplish this, a database trigger can be created that will automatically execute the above statement whenever there is an update to the tuple.

Once the Images are loaded and the features extracted, the *Resembles* function is used to retrieve those images similar to a given image. The *Resembles* function accepts a number of parameters:

- The database image and query feature vectors to be compared.
- A real number between 0 and 1 that is a cut off threshold in the similarity score. Only images that match with a score higher than the threshold are returned. We refer to such a cutoff as the *alpha cut* value.
- A weighting value for each of the features used. The weights do not have to add up to any particular value, but taken together, they cannot exceed 100. Weights are relative, so the weights (1,1,1,1,2,1) and (5,5,5,5,10,5) are equivalent.
- An output variable that contains the returned match score value.
- Query the *photo_collection* table with an example-image.

  The user provides an image feature vector as a query template. This feature vector can either be stored in the table, or correspond to an external image. Using a feature vector for an image already in the table requires a self join to identify the query feature vector. A feature vector for an external image requires calling the *GetFeatureVector* user-defined function.

  The first example uses an image already in the table (the one with image id 3) as the query image:

```
SELECT g.photo_id, score
       FROM photo_collection g, photo_collection s
       WHERE
           s.photo_id = 3
         AND
           Resembles(g.fv, s.fv, 0.0, 1, 1, 1, 0, 0, 0, score #REAL)
       ORDER BY score
```

  The *Resembles* function takes two extracted feature vectors (here *g.fv* and *s.fv*), computes a similarity score, and compares it to the indicated threshold. In this example, the threshold is 0.0, which means all images will be returned to the user. Following the threshold, six values in the argument list identify the weights for each of the features. Here, only the first three features (color, shape and texture) are used, while the remaining three are unused (their weights are set to 0). The last parameter is an output variable named *score* of type REAL, which contains the similarity score for the image match between the query feature vector *s.fv* and the images stored in the table. The score is then used to sort the result vectors to provide a ranked output.

The next example uses an external image as the query image with all features used for matching, and a non-zero threshold specified:

```
SELECT photo_id, score
       FROM photo_collection
       WHERE Resembles(fv,
              GetFeatureVector(IfdImgDescFromFile('/tmp/03.jpg')),
              0.80, 10, 30, 40, 10,
              5, 5, score #REAL)
       ORDER BY score
```

Note how the features are extracted in-situ by the *GetFeatureVector* function and passed to the *Resembles* function to compute the score between each image and the query image. In this query, only those images with a match score greater than 0.8 will be returned.

4.2 DB2 UDB Image Extender

IBM offers a full content management suite that, like the Media Asset Management Suite of Informix, provides a range of content administration, delivery, privilege management, protection, and other services. The IBM Content Manager product has evolved over a number of years, incorporating technology from several sources including OnDemand, DB2 Digital Library, ImagePlus, and VideoCharger. The early focus of these products was to provide integrated storage for and access to data of diverse types (i.e., scanned handwritten notes, images, etc.). These products, however, only provided search based on meta-data. For example, searching was supported on manually entered attributes associated with each digitized image, but not on the image itself. This, however, changed with the conversion of the IBM QBIC[7] prototype image retrieval system into a DB2 Extender. DB2 now offers integrated image search from within the database via the DB2 UDB Image Extender, which supports several color and texture feature representations.

In order to build the image retrieval application using the Image Extender, the following tasks need to be performed:

1. Install DB2 and the Image Extender, and configure the necessary storage space for the server. This installs a number of extender supplied user-defined distinct types and functions.

2. Create a Database to store all tables and auxiliary data needed for our example. We will call this the *Gallery* database.

   ```
   CREATE DATABASE Gallery;
   ```

[7]QBIC [38], standing for *Query By Image Content*, was the first commercial content-based Image Retrieval system and was initially developed as an IBM research prototype. Its system framework and techniques had profound effects on later Image Retrieval systems. QBIC supports queries based on example images, user-constructed sketches and drawings and selected color and texture patterns. The color features used in QBIC are the average (R,G,B), (Y,i,q),(L,a,b) and MTM (Mathematical Transform to Munsell) coordinates, and a k element Color Histogram. Its texture feature is an improved version of the Tamura texture representation [108], i.e., combinations of coarseness, contrast and directionality. Its shape feature consists of shape area, circularity, eccentricity, major axis orientation and a set of algebraic moments invariants.

3. Enable the *Gallery* database for Image searches. From the command line (not the SQL interpreter) use the Extender manager and execute:

   ```
   db2ext ENABLE DATABASE Gallery FOR DB2IMAGE
   ```

   This example uses the DB2 UDB version for UNIX and Microsoft Windows operating systems.

4. Create a table with the desired fields:

   ```
   CREATE TABLE photo_collection (
                photo_id integer PRIMARY KEY NOT NULL,
                photographer varchar(50) NOT NULL,
                date varchar(12) NOT NULL,
                photo DB2IMAGE)
   ```

5. Enable the table *photo_collection* for content-based image retrieval. This step again uses the external Extender manager, and is composed of several substeps.

   - Set up the main table, create auxiliary tables and indexes.

     ```
     db2ext ENABLE TABLE photo_collection FOR DB2IMAGE USING TSP1,,LTSP1
     ```

     This creates some auxiliary support tables used by the Extender to support image retrieval for the *photo_collection* table. These tables are stored in the database table-space named "*TSP1*" while the supporting large objects (BLOBs) are stored in the "*LTSP1*" table-space. The necessary indexes on auxiliary tables are also created in this step.

   - Enable the *photo* column for content-based image retrieval. This step again uses the external Extender manager.

     ```
     db2ext ENABLE COLUMN photo_collection photo FOR DB2IMAGE
     ```

     This makes the *photo* column active for use with the Image Extender and creates triggers that will update the auxiliary administrative tables in response to any change (insertion, deletion, update) to the data in table *photo_collection*.

   - Create a catalog for querying the column by image content. This is done with the extender manager.

     ```
     db2ext CREATE QBIC CATALOG photo_collection photo ON
     ```

     This creates all the support tables necessary to execute a content-based image query. The keyword *ON* indicates that the cataloging process (i.e., the feature extraction) will be performed automatically, otherwise, periodic manual re-cataloging is necessary.

   - Open a catalog for adding features, for which feature extraction is to take place; only those features present in the catalog will be available for querying. Using the Extender manager, we issue the following command.

     ```
     db2ext OPEN QBIC CATALOG photo_collection photo
     ```

   - Add those features for which feature extraction should take place to the catalog. Here we will add all four supported features.

     ```
     db2ext ADD QBIC FEATURE QbColorFeatureClass
     db2ext ADD QBIC FEATURE QbColorHistogramFeatureClass
     db2ext ADD QBIC FEATURE QbDrawFeatureClass
     db2ext ADD QBIC FEATURE QbTextureFeatureClass
     ```

     These correspond to *Average Color*, *Histogram Color*, *Positional Color*, and *Texture*.

Not all features need to be present, including unnecessary features will only decrease performance.

- Close the catalog.

```
db2ext CLOSE QBIC CATALOG
```

6. Insert into the *photo_collection* table. The examples presented here use *embedded* SQL to access a DB2 database server.

```
EXEC SQL BEGIN DECLARE SECTION;
  long int_Stor;
  long the_id;
EXEC SQL END DECLARE SECTION;

  the_id = 1; /* the image id */
  int_Stor = MMDB_STORAGE_TYPE_INTERNAL;

EXEC SQL INSERT INTO photo_collection VALUES(
                :the_id, /* id */
                'Ansel Adams', /* name */
                '6/9/2000', /* date */
                DB2IMAGE( /* Image Extender UDF*/
                  CURRENT SERVER, /* database server name */
                  '/images/pic.jpg' /* image source file*/
                  'ASIS', /* keep image format*/
                  :int_Stor, /* store in DB as BLOB*/
                  'BW Picture') /* comment*/
                );
```

This insert populates the image data in the auxiliary tables and stores an image handle into the *photo_collection* table. The DB2IMAGE user-defined function uses the current server, reads the image located in */images/pic.jpg*, and stores it in the server as specified by the *int_Stor* variable. The image is stored without a format change, and the discovery of the image format is left to the Image Extender, this is specified by the *ASIS* option. Features are extracted and stored for the image. The comment BW picture is attached to the image in the auxiliary tables. The DB2IMAGE user-defined function offers several different parameter lists (that is, it is an overloaded function), to support different sources to import images.

7. Query the *photo_collection* table with an example-image.

```
SELECT T.photo_id, T.photographer, S.SCORE
       FROM photo_collection T,
            TABLE (QbScoreTBFromStr(
                   'QbColorFeatureClass color=<255,0,0> 2.0 and
                    QbColorHistogramFeatureClass file=<server,"/img/pic1.gif"> 3.0 and
                    QbDrawFeatureClass file=<server,"/img/pic1.gif"> 1.0 and
                    QbTextureFeatureClass file=<server,"/img/pic1.gif"> 0.5',
                   photo_collection,
                   photo,
                   100)
             ) AS S
       WHERE CAST(S.IMAGE_ID as varchar(250)) = CAST(T.photo as varchar(250))
```

This query uses the image stored in */img/pic1.gif* as a query image and uses all four features. The *QbScoreTBFromStr* user-defined function takes a query string, an enabled table (*photo_collection*), a column (*photo*) name, and a maximum number of images to return.

This user-defined function returns a table with two columns. The first column is named *IMAGE_ID* and contains the image handle used by the Image Extender in the original table (.e., table *photo_collection*). The second column is named *SCORE* and is a numeric value, which denotes the query to image similarity score interpreted as a distance. A score of 0 denotes a perfect match and higher values indicate progressively worse matches.

The query string is structured as an *and* separated chain of *feature name*, *feature value*, *feature weight* triplets. The *feature name* indicates which feature to match. The *feature value* is a specification of the value for the desired feature and can be specified in several ways: (1) literally specifying the values, which is cumbersome as it requires that the user know the internal representation of each feature, (2) an image handle returned by the image Extender itself so an already stored image can be used as the query, and (3) an external file, for which the features are extracted and used. The above example uses the first approach for the average color feature, specifying an average color of red. The remaining three features use the third approach and use an external image, from which features are extracted for the query. The *feature weight* indicates the weight for this feature and is relative to the other features — if a weight is omitted, then a default value of 1.0 is assumed.

The table returned by the *QbScoreTBFromStr* user-defined function is joined on the image handle with the *photo_collection* table to retrieve the *photo_id* and *photographer* attributes and keep the score of the image match with the query.

4.3 Oracle Visual Image Retrieval Cartridge

Like Informix and IBM, Oracle supports a comprehensive media management framework named Oracle Intermedia that incorporates a number of technologies. Oracle Intermedia is designed with the objective of managing diverse media by providing many services from long term archival, to content-based search of text and images, to video storage and delivery. The Oracle Intermedia media management suite [75] contains a number of products designed to manage rich multimedia content particularly in the context of the web. Specifically, it includes components to handle audio, image and video data types. A sample application for this product would be an online music store that wishes to offer music samples, photos of the CD cover and performers, and a sample video of the performers. Intermedia is a tool box that includes a number of object-relational datatypes, indices, etc. that provide storage and retrieval facilities to web servers and other delivery channels including streaming video and audio servers. The actual media data can be stored in the server for full control, or externally, without full transactional support in a file system, web server, streaming server or other user-defined source. Functions implemented by this suite include among others, dynamic image transcoding to provide both thumbnails and full resolution images to the client upon request. As part of the Intermedia suite, the Oracle Visual Image Retrieval (VIR) product supplied by Virage [6, 44, 76][8] provides image search capabilities.

VIR supports matching on global color, local color, texture, and structure. Global color captures the images global color content, while local color takes into account the location of the color in the image. Texture distinguishes different patterns and nuances in images such as smoothness or graininess. Structure seeks to capture the overall layout of a scene such as the horizon in a photo, or the tall vertical boxes of skyscrapers. The product supports arbitrary combinations of the supported feature representations as a query. Users can adjust the weights associated with the features in the query according to the aspects they wish to emphasize. A score that incorporates the matching of all features is computed for each image via a weighted summation of the individual

[8]Virage also provided a version of its image retrieval system to Informix and is supported as a Datablade.

feature matches. The score is akin to the distance between two images where lower (positive) values indicate higher similarity, while larger values indicate lower similarity. Only those images with a score below a given threshold are returned, and the remaining images are deemed not relevant to the query. Oracle Visual Image Retrieval uses a proprietary index to speed up the matching referred to as an index of type *ORDVIRIDX*.

We now specify the steps needed to build an image retrieval application. The example code presented below uses Oracles PL/SQL language extensions. PL/SQL is a procedural extension to SQL. To support image retrieval the following steps are required:

1. Have Oracle8i Enterprise Edition and the Visual Image Retrieval product installed and suitably configured storage table-spaces.

2. Create a Database to store all tables and auxiliary data needed for our example. We will call this the *Gallery* database.

```
CREATE DATABASE Gallery;
```

3. Create a table with the desired attributes and the image datatype.

```
CREATE TABLE photo_collection (
             photo_id number PRIMARY KEY NOT NULL,
             photographer VARCHAR(50) NOT NULL,
             date VARCHAR(50) NOT NULL,
             photo ORDSYS.ORDVir);
```

4. Insert images into the newly created table. In Oracle this will be done through their PL/SQL language as there are multiple steps to insert an image.

```
DECLARE
    image ORDSYS.ORDVIR;
    the_id NUMBER;
BEGIN
    the_id :=1; -- use a serial number
    INSERT INTO photo_collection VALUES (
           the_id, 'Ansel Adams', '03/06/1995',
           ORDSYS.ORDVIR(ORDSYS.ORDImage(ORDSYS.ordsource
             (empty_BLOB(), 'FILE', 'ORDVIRDIR', 'the_image.jpg', sysdate, 0),
             NULL, NULL, NULL, NULL, NULL, NULL, NULL), NULL) );
    SELECT photo INTO image
           FROM photo_collection
           WHERE photo_id = the_id
           FOR UPDATE;
    image.SetProperties;
    image.import(NULL);
    image.Analyze;
    UPDATE photo_collection
           SET photo = image
           WHERE id = the_id;
END
```

The insert command only stores an image descriptor, not the image itself. To get the image, first its properties have to be determined using the `SetProperties` command. Then the image itself is loaded in with the `import(NULL)` command and its features extracted with the `Analyze` command. Lastly the table is updated with the image and its extracted features.

5. Create an index on the features to speed up the similarity queries.

```
CREATE INDEX imgindex
       ON catalog_photos(photo.signature)
       INDEXTYPE IS ordsys.ordviridx
       PARAMETERS ('ORDVIR_DATA_TABLESPACE = tbs_1,
       ORDVIR_INDEX_TABLESPACE = tbs_2');
```

   Here *tbs_1* and *tbs_2* are suitable table-spaces that provide storage.

6. Query the *catalog_photos* table.

   The following example selects images that are similar to an image already in the table with id equal to 3.

```
SELECT T.photo_id, T.photo, ORDSYS.VIRScore(50) SCORE
       FROM catalog_photos T, catalog_photos S
       WHERE
           S.photo_id = 3
         AND
           ORDSYS.VIRSimilar(T.photo.signature, S.photo.signature,
            'globalcolor="0.2" localcolor="0.3" texture="0.1" structure="0.4" ',
            20.0, 50)=1;
```

   This statement returns three columns, the first one is the *id* of the returned image, the second column is the image itself, and the third column is the score of the similarity between the query image and the result image (the parameter to the *VIRScore* function is discussed below). The query does a self join to fetch the value *S.photo.signature* for the image with an *id* of 3, which is the signature of the query image. The image similarity computation is performed by the *VIRSimilar* function in the query condition. This function has five arguments:

   - T.photo.signature, the compared images features.
   - S.photo.signature, the query image features.
   - A string value that describes the features and weights to be used in matching. This example has the string:
     `'globalcolor="0.2" localcolor="0.3" texture="0.1" structure="0.4" '`
     The value 0.0 for a weight indicates the feature is unimportant and the value 1.0 indicates the highest importance for that feature. Only those features listed are used for matching. If, for example, global color is not needed, then it may be removed from the list. In this example, all features are used and their weights are 0.2 for global color, 0.3 for local color, 0.1 for texture and 0.4 for structure.
   - The fourth parameter is a threshold for deciding which images are similar enough to the query signature to be returned for the query. The Image Retrieval Cartridge uses a distance interpretation of similarity. A score of 0 indicates the signatures are identical, while scores higher than 0 indicate progressively worse matches. In this example, the threshold value is 20.0, i.e., those images with a score larger than 20.0 will not be returned in response to the query.
   - The last value is optional and is used to recover the computed similarity score. The alert reader may have noticed that the *VIRSimilar* function is in a *where* clause, a Boolean condition, and therefore must return true or false, as opposed to the computed similarity score. The function returns true if the computed score is below the threshold, and false otherwise. If the query wishes to list for each retrieved image its similarity score to the

query, as is the case here, a different mechanism is needed to retrieve the score elsewhere in the query. This parameter value is thus used to uniquely identify the similarity score (computed by the *VIRSimilar* function) within the query in order to make it available elsewhere in the query through the use of the *VIRScore* function. *VIRScore* retrieves the similarity score by providing the same number as in the *VIRSimilar* function. This key-based identification mechanism enables multiple calls to scoring functions within the same query.

The final step in the query is to sort the result in increasing order of SCORE such that the most similar image will be the first one returned.

This example uses an image already in the table as the query image, but an external image may also be used. To do this, extra steps are needed similar to the *insert* command where an external image is read in and its features extracted and used in the *VIRSimilar* function. This scenario does not require a self join as the query feature vector is directly accessible.

Additional functionality is provided by a third-party software package from Visual Technology. This component supports special-purpose operators for searching for human faces among images stored in the database . Besides image search, the Visual Image Retrieval package offers a number of additional operational options such as image format conversion and on-demand image transcoding of query results.

4.4 Discussion

We have discussed the extensions supported for incorporating images and multimedia into databases by three of the major DBMS vendors. All the vendors discussed offer media asset management suites to archive and manage digital media. Their offerings differ in the details of their composition, scope and source (i.e., third party vs. home grown) and their maturity. The image retrieval capabilities of all vendors are roughly comparable. Despite minor administrative differences in table and column setup, once the tables and permissions are set properly, the insertion and querying process is comparable. Each of the image retrieval products discussed above essentially supports the base content-based image retrieval model discussed in Section 2. There is, however, one difference.

Recall that in Section 2, the model permits several query-example images, but so far in this section, we only considered single-example-image queries. Multiple example-image query support is beyond the current query model implemented by these vendors, but is not impossible to implement. Indeed, the model can be incorporated in a query, albeit in an exposed fashion. Exposed, because now the user writing the query is exposed to the retrieval model and is responsible for formulating a query properly. To see how such a query can be specified, we will use Informix as an example:

```
SELECT photo_id, (score1 * 0.6 + score1 * 0.4) as score
       FROM photo_collection
       WHERE Resembles(fv,
               GetFeatureVector(IfdImgDescFromFile('/img/query1.jpg')),
               0.60, 10, 30, 40, 10, 5, 5, score1 #REAL)
         AND Resembles(fv,
               GetFeatureVector(IfdImgDescFromFile('/img/query2.jpg')),
               0.60, 20, 20, 20, 5, 5, 5, score2 #REAL)
       ORDER BY score
```

This query uses two external images, *query1.jpg* and *query2.jpg* and computes the score between each individual image in the table and the *query1.jpg* and *query2.jpg* image feature vectors *fv* resulting in one score for each of the two example-images. Then it combines both scores with a weighted

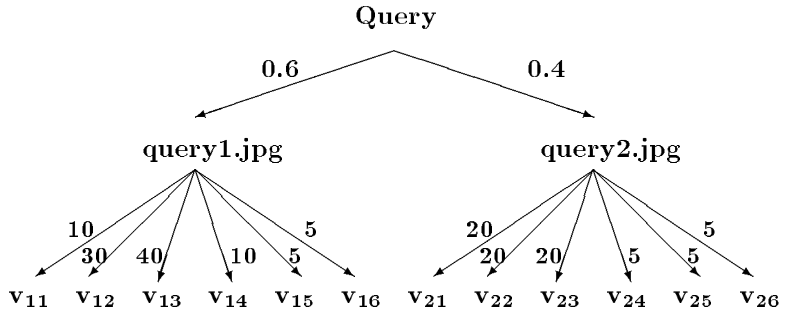

Figure 2: Query example

summation with 60% of weight for *query1.jpg* and 40% of the weight to *query2.jpg*. Notice that both *Resembles* function calls specify a threshold of 0.60 and that they use different weights for different features. Figure 2 shows the query tree that corresponds to this example. In this figure, the leaf nodes correspond to actual values v_{ij} for the query image i and the feature j.

It is not clear if the feature-based model described in Section 2 can be supported by existing systems. Furthermore, we note that none of the products currently available is powerful enough to support region-based image retrieval, relevance feedback mechanisms, or "merge" functions other than weighted summation at different levels of the query tree. Extending image retrieval with these functionalities is a significant research challenge, which the research community has yet to address.

Finally, we note that the above description of the image extensions to commercial DBMSs is certainly not complete. Besides content-based image retrieval, products include many functions that are designed to handle operational considerations, such as image format conversions and image processing. Interested readers are referred to the product manuals.

5 Current Research

The advent of object-relational technology has greatly facilitated the building of content-based image retrieval applications on top of commercial database systems. Using ADTs, UDFs and BLOBs supported by commercial systems, applications can extract the visual features of images, store the features in relational tables (along with the raw images themselves), and use these features to compute query matches. While this represents significant progress, several proposals to improve the above approach have appeared in the research literature. One of them is an efficient technique to implement abstract data types on top of relational storage managers. Another issue is that of allowing users to easily integrate multidimensional indexing structures into the DBMS, and use them as access methods. This will allow applications to build indexes on the image features and use them to efficiently answer content-based queries. To realize the full potential of the feature indexes, the processing of content-based queries (i.e., top-k and threshold-based queries) must be pushed inside the database engine. Commercial systems, such as those described in Section 4, retrieve all the matching objects from the database with their computed scores and then perform most of the processing (i.e., sorting and pruning) as an independent step. This misses out on opportunities for optimization and efficient evaluation of content-based queries. Pushing the processing into the engine would open up such optimization opportunities, leading to tremendous performance gains. Another proposal is that of efficient support for similarity joins to facilitate finding similar pairs of images.

5.1 Implementing ADTs using Relational Storage Managers

While abstract data types (ADTs) have appeared in mainstream commercial databases [74, 22, 53, 107], they present several challenges in terms of storage management. Abstract data types support varied functionalities, such as inheritance, polymorphism, substitutability, encapsulation, structures, and collections among others. We discuss the storage management problems that arise when an ADT is defined as an aggregation of base data types and/or already defined ADTs (like the way structs are defined in C). In our discussion, we do not consider "opaque" types where the system treats the type as an (uninterpreted) chunk of memory, which is interpreted by the user-defined functions [53]. We also do not consider the functions defined for the ADT here but will cover them in more detail in Section 5.3.1.

Consider an ADT for a few geometric shapes in two dimensions:[9]

```
Type 2dShape ( )
Type Point inherits from 2dShape ( x,y :integer)
Type Circle inherits from 2dShape ( x,y, radius :integer)
Type Rectangle inherits from 2dShape ( x1,y1, x2,y2 :integer)
Type Triangle inherits from 2dShape ( x1,y1, x2,y2, x3,y3 :integer)
```

Suppose we want to associate one or more regions with each image in our *Gallery* database from Section 4. The idea is to create image maps for the users to click on. Assuming that each region is one of the above 2d shapes,[10] we now create a table to store the regions.

```
create table photo_regions(region_id integer, photo_id integer, region 2dShape)
```

This table would allow a region to be a point, a circle, a rectangle or a triangle by virtue of type substitutability. Now, we add the following data to our table:[11]

```
insert into photo_regions values (1, 1, Point (100,100))
insert into photo_regions values (2, 1, Circle (10,10, 5))
insert into photo_regions values (3, 1, Rectangle(50,60, 80,90))
insert into photo_regions values (4, 1, Triangle (0,0, 5,5, 5,0))
```

Here, we insert a point located at the coordinates (100,100), a circle of radius 5 centered at the point (10,10), a rectangle with opposing corner points (50,60) and (80,90) and a triangle with the three corners (0,0), (5,5) and (5,0). Each tuple above stores one integer for the *region_id*, one integer for the *photo_id* plus an internal fixed sized tuple header whose size depends on the DBMS used. The rest of the information in all four tuples differs from each other: the first tuple needs to store two more integers (in addition to header, region_id and photo_id), the second needs three more, the third needs four more and the fourth needs to store six more integers. The question is how to organize the above tuples on disk in order to handle the diversity among them without sacrificing query efficiency. The following options are available:

1. *One table for each possible data type*

   Within this scheme, although there is only one logical table *photo_regions*, the system creates 5 physical tables, one for each of 2dShape, Point, Circle, Rectangle, and Triangle.[12] Each

[9]Here we have used generic SQL pseudo-code. For specific vendor implementations and syntax, the appropriate manual should be consulted.

[10]More flexible shapes, such as polygons are necessary for such an application. Here we restrict ourselves to the above shapes as our goal is to show the problems faced in storage management of ADTs, and not to provide a complete image mapping solution.

[11]We assume that the appropriate object constructors have been defined (i.e., "Point(x,y)" has been defined as a constructor for the Point type).

[12]There should be no table for 2dShape itself assuming it is a pure abstract data type whose only purpose is to

table has a uniform and fixed schema and it is up to the query processor to look in all the physical tables for a query on the logical table. The advantages of this approach are that regular relational storage managers can be used, the tuples have fixed length and are therefore more amenable to optimizations, and, since the schema for each physical table is known in advance, no dynamic interpretation of object types is needed. The disadvantages are that the query processor must decide which tables to search for a query and, on some occasions, it might be necessary to search all the physical tables. More importantly, there could be an explosion in the number of physical tables if the inheritance hierarchy is deep. The Postgres and Illustra systems used an approach similar to this one [103, 104].

2. *Co-locate tuples of different types in one table*

   In this approach, a single physical table is used to store all the five different types of tuples. Like approach 1, almost no dynamic decoding of the object type is required, as the layout and column information of each tuple type is fully pre-computed. Another advantage is that all the tuples are stored in the same table avoiding the need for multiple-table lookups. The disadvantage is that there could be an explosion of the number of tuple types in the table. This approach is used in the MARS file system.

3. *Flatten ADT and map to regular relation*

   In this approach, all the tuples are stored in a single table, which is managed by a regular relational storage manager (i.e., the storage manager need not support multiple tuple types per relation). All the types are expanded into their components and stored in individual columns of the table. The columns that are not used are filled with NULL values. In this approach, the *photo_regions* table would have 17 columns (region_id, photo_id, 2 columns for Point, 3 for Circle, 4 for Rectangle and 6 for Triangle)[13] and most of them will contain NULL values as only those columns that correspond to the actual object stored will be non NULL (i.e., only 4 columns for a Point object would be non NULL and the remaining 13 would be NULL). The advantage of this approach is that it can be readily implemented in regular relational storage managers. The disadvantages are that space is wasted and additional work is needed in the query processor to dynamically interpret the correct columns for a tuple. Note that, as in the previous approaches, there could be an explosion in storage requirements, since here the number of columns needed may increase rapidly.

4. *Serialize object in-line and dynamically interpret content to determine type*

   In this approach, the table schema is exactly as desired with three attributes, one each for region_id, photo_id and 2dShape. The last attribute is now stored as a variable length column, either in-line in the tuple, or out of line in a BLOB if its size is too large. This approach has several advantages. It is the most flexible since it can most easily handle changes in the type hierarchy (the other three approaches must make significant changes in the schema of the table(s) when the type hierarchy changes.) It also avoids the combinatorial explosion and space overhead problems of the previous approaches. The only disadvantage is the overhead of dynamically interpreting the contents of each tuple to determine its type. IBM DB2 follows a similar approach to store ADTs [39]. Sybase also follows a similar approach for Java objects [106].

serve as the common superclass, i.e., there can be no instantiations of this type. However, here we have included 2dShape as a normal ADT.

[13] In practice, one more column is needed to keep track of which object is stored in the table.

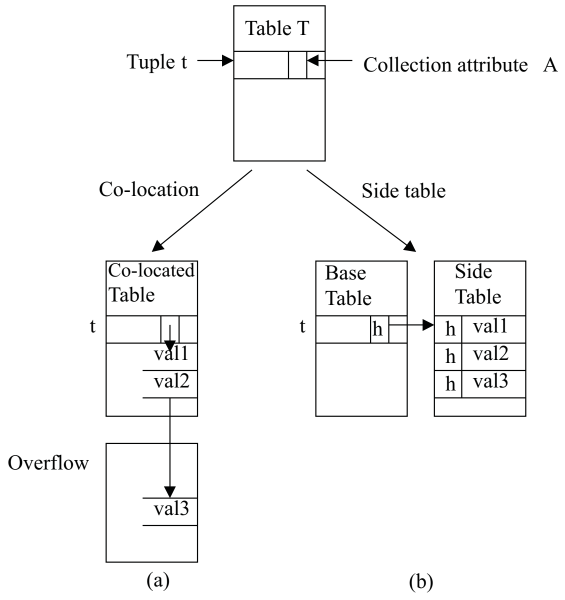

Figure 3: Two ways of implementing collections inside attributes. (a) The co-location approach (b) The side table approach.

A study on the implementation of ADTs comparing several variations of approaches 3 and 4 can be found in [39]. Storage management of rich data types has also been addressed by object-oriented databases [7, 66].

Another important problem in ORDBMSs that is relevant to image retrieval is the management of collection-type attributes. An example of such an attribute is the polygonal contour shape descriptor (represented by the corner points, the number of which can vary from shape to shape). Such attributes are usually handled by co-location or by using side tables. In the co-location approach, the items in the collection attribute of a tuple are stored along with the rest of the tuple with an optional pointer to an overflow area. Figure 3(a) shows how the items $\{val1, val2, val3\}$ in the collection attribute A of a tuple t would be stored. The advantage of this approach is efficiency, as most of the time, all the items in the collection will be co-located with the tuple (i.e., no overflow pointer required). The disadvantages are that updates to a tuple may cause the use of the overflow areas, thus increasing fragmentation and degrading performance. Also, the storage manager must be able to support coexistence of tuples from different schemas in the same file. In the side table approach, there is a base table and there is a separate side table for each column with a collection attribute. Each tuple t in the base table stores a system generated handle h for each collection attribute A. The handle h is a key into a side table where a tuple $\langle h, item\rangle$ is stored for each item $item$ in the collection attribute A of tuple t. Figure 3(b) shows how the same table t with items $\{val1, val2, val3\}$ in the collection attribute A is stored using this approach. An advantage of this approach is that it does not require any special support from storage managers, as all tuples in a relation are the same. Fragmentation in the side table can also be avoided by having the file clustered on the handle. A clear disadvantage is that potentially expensive joins or table lookups are needed.

5.2 Multidimensional Access Methods

As discussed in Section 2, image retrieval systems represent the content of the images using visual features like color, texture and shape. Processing content-based queries on large image collections can be speeded up significantly by building indices on the individual features (known as the feature indices or simply F-indices) and using them to answer content-based queries. Since the feature spaces are high-dimensional in nature (i.e., 32-dimensional color histogram space), novel indexing techniques must be developed and incorporated into the DBMS (cf. Chapters 14 and 15). The purpose of a feature index is to efficiently retrieve the best matches with respect to that feature by executing a range search or a k-nearest neighbor (k-NN) search on the multidimensional index structure. How these individual feature matches returned by the feature indices can be used to obtain the overall best matches will be discussed in Section 5.3.1. In this section, we discuss research issues that arise in designing index structures that can execute range and k-NN searches efficiently over image feature-spaces, in supporting new types of queries for image retrieval applications, and in integrating into the DBMS multidimensional index structures.

5.2.1 Designing Index Structures for Image Feature Spaces

The main problem that arises in indexing image feature spaces is high dimensionality. For example, the color histograms used in the MARS system are usually 32- or 64-dimensional [20]. Many multidimensional index structures do not scale to such high dimensionalities [10]. Designing scalable index structures has been an active area of research and is discussed in detail in Chapters 14 and 15.

5.2.2 Supporting Multimedia Queries on top of Multidimensional Index Structures

Traditionally, multidimensional index structures support only point, range and k-NN queries (with a single query point) and only the Euclidean distance function. In multimedia retrieval, the retrieval model defines what a match between two images means with respect to each individual feature. The measure of match (rather, mismatch) is defined in terms of a distance function. The retrieval model uses arbitrary distance functions (typically an L_p metric) and arbitrary weights along the dimensions of the feature space to capture the visual perception of the user. This implies that the index structure must support arbitrary distance functions and arbitrary weights along the dimensions that are specified by the user at query time. Such techniques have been developed in [62, 93, 21]. Another requirement of the index structure is to support multi-example queries, since the user might submit multiple images as part of the query (see section 2). Such queries are particularly important for retrieval models that represent the query using multiple query points [88, 110]. A multipoint query Q_F for a feature F is formally defined as $Q_F = \langle n_F, P_F, W_F, D_F \rangle$ where n_F is the number of points in the query, $P_F = \{{P_F}^{(1)}, ..., {P_F}^{(n_F)}\}$ is the set of n_F points in the feature space, $W_F = \{{w_F}^{(1)}, ..., {w_F}^{(n_F)}\}$ are the corresponding weights and D_F is a distance function, which, given two points in the feature space, returns the distance between them (usually a weighted L_p metric). The distance between the multipoint query Q_F and an object point O_F with respect to feature F is defined as the aggregate of the distances between O_F and the individual points ${P_F}^{(i)} \in P_F$ in Q_F. The weighted sum

$$D_F(Q_F, O_F) = \sum_{i=1}^{n_F} {w_F}^{(i)}\, D_F({P_F}^{(i)}, O_F) \tag{2}$$

may be used as an aggregation function. The individual point-to-point distance $D_F({P_F}^{(i)}, O_F)$ is given by a weighted L_p metric

$$\mathcal{D}_F({P_F}^{(i)}, O_F) = \left[\sum_{j=1}^{d_F} {\mu_F}^{(j)} \left(|{P_F}^{(i)}[j] - O_F[j]| \right)^p \right]^{1/p}, \tag{3}$$

where d_F is the dimensionality of the feature space and ${\mu_F}^{(j)}$ denotes the (*intra-feature*) weight associated with the jth dimension. This is the aggregation function used in the MARS system. The problem is to find the k nearest neighbors of Q_F using the F-index.

One way to implement a multipoint query is the multiple expansion approach proposed by Porkaew et al. [89] and Wu et al. [110]. The approach explores the nearest neighbors of each individual point ${P_F}^{(i)}$ using the traditional single-point k-NN algorithm and combines them. An alternate way, proposed in [88, 21], is to develop a new k-NN algorithm that can handle multipoint queries. The latter technique involves (1) redefining the MINDIST function, which is used to compute the distance of an index node from the query and (2) using the distance function described above to compute the distance of a indexed object from the multipoint query. Experiments show that the latter technique can process a multipoint query much more efficiently compared to the multiple expansion approach [21].

5.2.3 Integration of Multidimensional Index Structures as Access Methods in a DBMS

While there exists several research challenges in designing scalable index structures and developing algorithms for efficient content-based search using them, one of the most important practical challenges is that of integration of such indexing mechanisms as access methods (AMs) in a database management system. Building a database server with native support for all possible kinds of complex multimedia features and the feature-specific indexing mechanisms along with support for feature-specific queries/operations is not feasible. The solution is to build an extensible database server that allows the application developer to define data types and related operations as well as indexing mechanisms on the stored data, which the database query optimizer can exploit to access the data efficiently. As discussed in Section 3.3, commercial ORDBMSs have started providing extensibility options for users to incorporate their own index structures. As pointed out earlier, the interfaces exposed by current commercial systems are too low-level, and place the burden of writing structural maintenance code (i.e., concurrency control) on the access method implementor. The *Generalized Search Tree* (GiST) [47] provides a more elegant solution to the above problem.

Generalized Search Tree (GiST): A GiST is a balanced tree of variable fanout between kM and M, $\frac{2}{M} \leq k \leq \frac{1}{2}$, with the exception of the root node, which may have fanout between 2 and M. The constant k is termed the minimum fill factor of the tree. Leaf nodes in a GiST contain (p, ptr) pairs, where p is a predicate that is used as a search key and ptr is the identifier of some tuple in the database. Non-leaf nodes contain (p, ptr) pairs, where p is a predicate used as a search key (referred to as bounding predicate (BP) [19]) and ptr points to another tree node. The predicates can be arbitrary as long as they satisfy the following condition: the predicate p in a leaf node entry (p, ptr) must hold for the tuple identified by ptr while the bounding predicate p in a non-leaf node entry (p, ptr) must hold for any tuple reachable from ptr. A GiST for a key set comprised of 2-d rectangles is shown in Figure 4.

Generalizing the notion of a search key to an arbitrary predicate makes GiST extensible, both in the datatypes it can index and the queries it can support. GiST is like a "template" – the application developer can implement a new AM using GiST by simply registering a few (domain-specific)

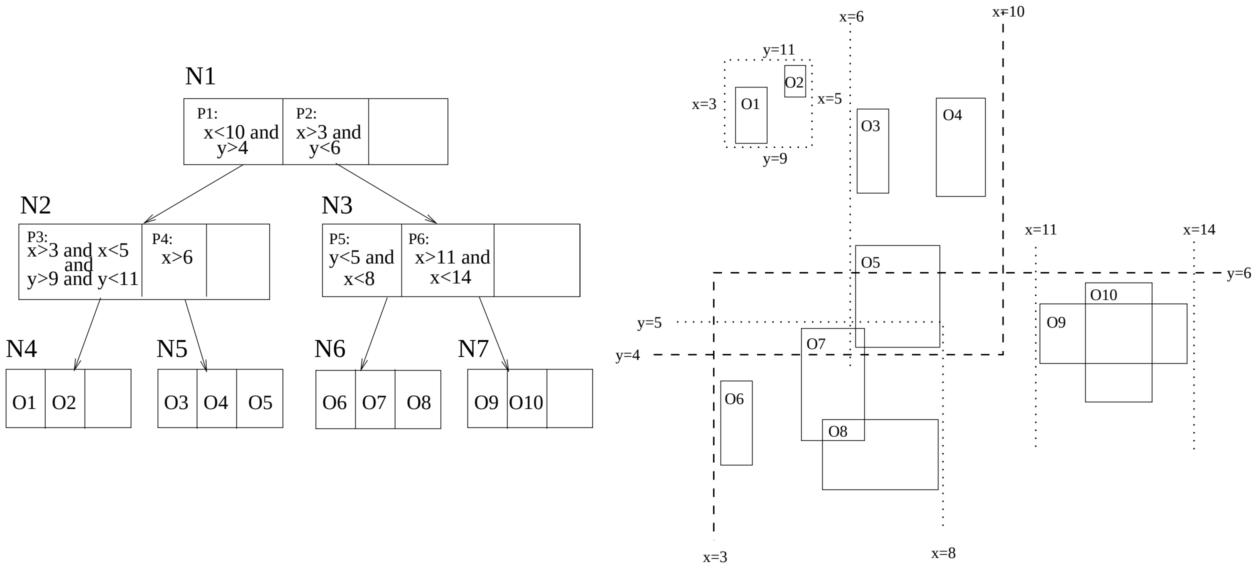

Figure 4: A GiST for a key set comprised of rectangles in 2 dimensional space. Note that the bounding predicates are arbitrary (i.e. not necessarily bounding rectangles as in R-trees).

extension methods with the DBMS. Examples of the extension methods are $Consistent(E, q)$, which, given an entry $E = (p, ptr)$ and a query predicate q, returns false if $p \wedge q$ can be guaranteed unsatisfiable, and true otherwise, and $Penalty(E_1, E_2)$, which, given two entries $E_1 = (p_1, ptr_1)$, $E_2 = (p_2, ptr_2)$, returns a domain-specific penalty for inserting E_2 into the subtree rooted at E_1. GiST uses the extension methods provided by the AM developer to implement the standard index operations: search, insertion and deletion. For example, the search operation uses $Consistent(E, q)$ to determine which nodes to traverse to answer the query while the insert operation uses $Penalty(E_1, E_2)$ to determine the leaf node in which to place the inserted item in. The AM developer thus controls the organization of keys within the tree and the behavior of the search operation, thereby specializing GiST to the desired AM. The original GiST paper deals only with range queries [47]. Several extensions to support more general queries (i.e., ranked/nearest neighbor queries) on top of GiST are proposed in [3].

Concurrency Control in GiST: Although GiST considerably reduces the effort of integrating new AMs in DBMSs, it does not automatically provide concurrency control. It is essential to develop efficient techniques to manage concurrent access to data via the GiST, before it can be supported by "commercial strength" DBMSs. Concurrent access to data via an index structure introduces two independent concurrency control problems:

- *Preserving consistency of the data structure* in the presence of concurrent insertions, deletions and updates.
- *Protecting search regions from phantoms.*

Techniques for concurrency control (CC) in multidimensional data structures and, in particular, GiST have been proposed recently [63, 19, 18]. Developing CC techniques for GiST is particularly beneficial since the CC code can be implemented once by the database developer - the end-user does not need to implement individual algorithms for each AM.

Preserving Consistency of GiST: We first discuss the consistency problem and its solution. Consider a GiST (configured as, say, and R-tree) with a root node R and two children nodes A and

B. Consider two operations executing concurrently on the R-tree: an insertion of a new key $k1$ into B and a deletion of a key $k2$ from B. Suppose the deletion operation examines R and discovers that $k2$, if present, must be in B. Before it can examine B, the insertion operation causes B to split into B and B', as a result of which $k2$ moves to B' (and subsequently updates R). The delete operation now examines B and incorrectly concludes that $k2$ does not exist. To avoid the above problem, Kornaker et al. propose a linked-based technique that was originally used in B-trees [63]. By adding a right link between a node and its split-off right sibling and a node sequence number to every node, the operations (i.e., the deletion operation in the above example) can detect whether the node has split since the parent was examined and, if so, can compensate for the missed splits by following the right links.

Phantom Protection in GiST: We now move on to the problem of phantom protection. Consider a transaction T1 reading a set of data items from a GiST that satisfy some search predicate Q. Transaction T2 then inserts a data item that satisfies Q and commits. If T1 now repeats its scan with the same search predicate Q, it gets a set of data items (known as "phantoms") different from the first read. Phantoms must be prevented to guarantee serializable execution. Note that object-level locking [42] does not prevent phantoms since even if all objects currently in the database that satisfy the search predicate are locked, concurrent insertions[14] into the search range cannot be prevented. There are two general strategies to solve the phantom problem, namely *predicate locking* and its engineering approximation, *granular locking*. In predicate locking, transactions acquire locks on predicates rather than individual objects. Although predicate locking is a complete solution to the phantom problem, it is usually too costly [42]. In contrast, in granular locking, the predicate space is divided into a set of lockable resource granules. Transactions acquire locks on granules instead of on predicates. The locking protocol guarantees that if two transactions request conflicting-mode locks on predicates p and p' such that $p \wedge p'$ is satisfiable, then the two transactions will request conflicting locks on at least one granule in common. Granular locks can be set and released as efficiently as object locks. An example of the granular locking approach is the *multi-granularity locking protocol* (MGL) [68]. Application of MGL to the key space associated with a B-tree is referred to as *key range locking* (KRL) [68, 70].

In [63], Kornaker et al. develop a solution for phantom protection in GiSTs based on predicate locking. In the proposed protocol, a searcher attaches its search predicate Q to all the index nodes whose bounding predicates (BPs) are consistent with Q. Subsequently, the searcher acquires shared mode locks on all objects "consistent" with Q. An inserter checks the object to be inserted against all the search predicates attached to the node in which the insertion takes place. If it conflicts with any of them, the inserter attaches its predicate to the node (to prevent starvation) and waits for the conflicting transactions to commit. If the insertion causes a BP of a node N to grow, the predicate attachments of the parent of N are checked with the new BP of N, and are replicated at N if necessary. The process is carried out top-down over the entire path where node BP adjustments take place. Similar predicate checking and replication is done between sibling nodes during split propagation. The details of the protocol can be found in [63].

In [19], Chakrabarti and Mehrotra propose an alternative approach based on granular locking. Note that the granular locking technique for B-trees, viz., key range locking (KRL), cannot be applied for phantom protection in multidimensional data structures since it relies on a total order of key values, which does not exist for multidimensional data. Imposing an artificial total order (say a Z-order [80]) over multidimensional data to adapt KRL is not a viable technique either. The first step is to define lockable resource granules over the multidimensional key space. One way to

[14]These insertions may be a result of insertion of new objects, updates to existing objects or rolling-back deletions made by other concurrent transactions.

define the granules is to statically partition the key space (e.g., as a grid) and treat each partition (i.e., each grid cell) as a granule. The problem with such a partitioning is that it does not adapt to the key distribution: some granules may contain many more keys than others, causing them to become "hot spots". In [19], the authors use the predicate space partitioning generated by the GiST to define the granules. There is a lockable granule $TG(N)$ associated with each index node N of a GiST whose coverage is defined by the *granule predicate* $GP(N)$ associated with the node. $GP(N)$ is defined as follows. Let P denote the parent node of N (P is NULL if N is the root node), and $BP(N)$ denote the bounding predicate of N. The granule predicate $GP(N)$ of node N is equal to $BP(N)$ if N is the root and $BP(N) \wedge GP(P)$ otherwise. For example, the granule predicate associated with the non-leaf node $N2$ in Figure 4 is $P1 = (x < 10) \wedge (y > 4)$ while that associated with leaf node $N5$ is $P1 \wedge P4 = (6 < x < 10) \wedge (y > 4)$. The granules associated with leaf nodes are called leaf granules while those associated with non-leaf nodes are called non-leaf granules. Note that the above partitioning scheme does not suffer from the "hot spot" problem of static partitioning since the granules dynamically adapt to the key distribution as keys are inserted into and deleted from the GiST.

Once the granules are defined, the authors develop lock protocols for the various operations on the GiST. As mentioned before, for correctness, if two operations conflict, they must request conflicting locks on at least one granule in common. The protocol exploits, in addition to shared mode (S) and exclusive mode (X) locks, *intention* mode locks which represent the intention to set locks at finer granularity. The compatibility matrix for the various lock modes used by the protocol can be found in [19]. The lock protocol of the search operation is simple. A searcher acquires commit-duration S-mode locks on all granules (both leaf and non-leaf) "consistent" with its search predicate. Note that in this technique, unlike the approach of [63], the searcher does not acquire object-level locks. The lock protocol of the insertion operation is slightly more involved. Let O be the object being inserted and g be the granule corresponding to the leaf node in which O is being inserted. The protocol has the following two cases. If the insertion does not cause g to grow, the inserter acquires (1) a commit-duration IX-mode lock on g where the IX-mode is an intention mode (intention to set shared or exclusive mode locks at finer granularity) and (2) a commit-duration X-mode lock on O. Otherwise, the insertion acquires (1) a commit-duration IX-mode lock on g (2) a commit-duration X-mode lock on O and (3) a *short*-duration IX-mode lock on TG(LU-node) where the LU-node (Lowest Unchanged Node) denotes the lowest node in the insertion path whose GP does not change due to the insertion. The above protocol guarantees that a transaction cannot insert an object into the search region of another concurrently-running transaction, i.e., it will request a conflicting lock on at least one common granule and hence will block till the search transaction is over. The correctness proofs and the lock protocols of the other operations can be found [19].

5.3 Supporting Top-k Queries in Databases

In content-based image retrieval, almost all images match the query image to some degree or another. The user is typically not interested in all matching images (i.e., all images with degree of match > 0) as that might retrieve the entire database. Rather she is interested in only the top few matching images. There are two ways of retrieving the top few images: *Top-k* queries return the k best matches, irrespective of their scores. For example, in Section 4.2, we requested the top 100 images matching */image/pic1.gif*. *Range queries* or *alpha-cut* queries return all the images whose matching score exceeds a user-specified threshold, irrespective of their number. For example, in Section 4.1, we requested all images whose degree of match to the image */tmp/03.jpg* exceeds the alpha-cut of 0.8.[15] Database query optimizers and query processors do not support queries with user-specified limits on result cardinality; the limiting of the result cardinalities in the above examples (in Section 4) is achieved at the application level (by Excalibur Visual Retrievalware in Informix, QBIC in DB2 and Virage in Oracle). The database engine returns all the tuples that satisfy the non-image selection predicates, if any, and all tuples otherwise; the application then evaluates the image match for each returned tuple and retains only those that satisfy user-specified limit. This causes large amounts of wasted work by the database engine (as it accesses and retrieves tuples, most of which are eventually discarded) leading to long response times. Significant performance improvements can be obtained by pushing the top-k and range query processing inside the database engine. In this section, we discuss query optimization and query processing issues that arise in that context.

5.3.1 Query Optimization

Relational query languages, particularly SQL, are declarative in nature: they specify what the answers should be and not how they are to be computed. When a DBMS receives an SQL query, it first validates the query and then determines a strategy for evaluating it. Such a strategy is called the query evaluation plan or simply *plan* and is represented using an operator tree [40]. For a given query, there are usually several different plans that will produce the same result; they only differ in the amount of resources needed to compute the result. The resources include time and memory space in both disk and main memory. The query optimizer first generates a variety of plans by choosing different orders among the operators in the operator tree and choosing different algorithms to implement these operators, and then chooses the best plan based on the available resources [40]. The two common strategies to compute optimized plans are (1) *rule-based* optimization [46] and (2) *cost-based* optimization [94]. In the rule-based approach, a number of heuristics are encoded in the form of production rules that can be used to transform the query tree into an equivalent tree that is more efficient to execute. For example, a rule might specify that selections are to be pushed below joins, since this reduces the sizes of the input relations and hence the cost of the join operation. In the cost-based approach, the optimizer first generates several plans that would correctly compute the answers to a query and computes a cost estimate for each plan. The system maintains some running statistics for each relation (i.e., number of tuples, number of disk pages occupied by the relation, etc.) as well as for each index (i.e., number of distinct keys, number of pages etc.), which are used to obtain the cost estimates. Subsequently, the optimizer chooses the plan with the lowest estimated cost [94].

Access Path Selection for Top-k Queries: Pushing top-k query processing inside the database engine opens up several query optimization issues. One of them is access path selection. The access path represents how the top-k query accesses the tuples of a relation in order to compute the result. To illustrate the access path selection problem, let us consider an example image database where the images are represented using two features, color and texture. Assuming that all the extracted feature values are stored in the same tuple along with other image information (i.e., the photo_id, photographer and date in the example in Section 4), one option is to sequentially scan through the entire table, computing the similarity score for each tuple by first computing the individual feature scores and then combining them using the merge function, while retaining the k tuples with the highest similarity scores. This option may be too slow, especially if the relation is very large.

[15]For both types of queries, the user typically expects the answers to be ranked based on their degree of match (the best matches before the less good ones). For top-k queries, a "get more" feature is desirable so that the user can ask for additional matches if she wants.

Another option that avoids this problem is to index each feature using a multidimensional index structure. With the indexes in place, the optimizer has several choices of access paths:

- Filter the images on the color feature (using $k-$NN search on the color index), access the full records of the returned images, which contain the texture feature values, and compute the overall score.
- Filter the images on the texture feature, analyze the full records of the returned images, which contain the color feature values, and compute the overall score.
- Use both the color and texture indexes, to find the best matches with respect to each feature individually, and merge the individual results.[16]

Note the number of execution alternatives increases exponentially with the number of features. The presence of other selection predicates (i.e., the "date >= '01/01/2000'" predicate in the above example) also increases the size of the execution space. It is up to the query optimizer to determine the access path to be used for a given query. Database systems use a cost-based technique for access path selection as proposed by Selinger et al. [94]. To apply this technique, several issues need to be considered. In image databases, the features are indexed using multidimensional index structures, which serve as access paths to the relation. New kinds of statistics need to be maintained for such index structures and new cost formulae need to be developed for accurate estimation of their access costs. In addition, the cost models for top-k queries are likely to be significantly different from traditional database queries that return all tuples satisfying the user-specified predicates. The cost model would also depend on the retrieval model used, i.e., on the similarity functions used for each individual feature as well as the ones used to combine the individual matches (cf. Section 2.3). Such cost models need to be designed.

In [24], Chaudhari and Gravano propose a cost model to evaluate the costs of the various execution alternatives[17] and develop an algorithm to determine the cheapest alternative. The cost model relies on techniques for: estimating selectivity of queries in individual feature spaces; estimating the costs of k-NN searches using individual feature indices; and probing a relation for a tuple, to evaluate one or more selection predicates. Most selectivity estimation techniques proposed so far for multidimensional feature spaces work well only for low dimensional spaces, but are not accurate in the high-dimensional spaces commonly used to represent images features [87, 69, 1]. More suitable techniques (based, for instance, on fractals) are beginning to appear in the literature [8, 36]. Work on cost models for range and k-NN searches on multidimensional index structures includes earlier proposals for low dimensional index structures (i.e., R-tree) in [34, 109] and more recent work for higher dimensional spaces in [9, 84].

Optimization of Expensive User-Defined Functions: The system may need to evaluate multiple selection predicates on each tuple accessed via the chosen access path. Relational query optimizers typically place no importance on the order in which the selection predicates are evaluated on the tuples. The same is true for projections. Selections and projections are assumed to be zero-time operations. This assumption is not true in content-based image retrieval applications where selection predicates may involve evaluating expensive user-defined functions. Let us consider the following query on the `photo_collection` table in Section 4.1:

[16]Yet another option would be to create a combined index on color and texture features so that the above query can be answered using a single index. We do not consider this option in this discussion.

[17]The authors do not consider all the execution alternatives but only a small subset of them (called search-minimal executions) [24]. Also, the authors only consider Boolean queries and do not handle more complex retrieval models (i.e., weighted sum model, probabilistic model).

```
/* Retrieve all photographs taken since the year 2000 */
/* that match the query image more than 8%.  */
SELECT photo_id, score
       FROM photo_collection
       WHERE Resembles(fv,
              GetFeatureVector(IfdImgDescFromFile('/tmp/03.jpg')),
              0.80, 10, 30, 40, 10,
              5, 5, score #REAL)
       AND
              date >= '01/01/2000'
       ORDER BY score
```

The query has two selection predicates: the Resembles predicate and the predicate involving the date. The first one is a complex predicate that may take hundreds or even thousands of instructions to compute, while the second one is a simple predicate that can be computed in a few instructions. If the chosen access path is a sequential scan over the table, the query will run faster if the second selection is applied before the first, since doing so minimizes the number of calls to Resembles. The query optimizer must, therefore, take into account the computational cost of evaluating the UDF in order to determine the best query plan. Cost-based optimizers only use the selectivity of the predicates to order them in the query plan but do not consider their computational complexities. In [49, 48], Hellerstein et al. use both the selectivity and the cost of selection predicates to optimize queries with expensive UDFs. In [26], Chaudhari and Shim proposes dynamic-programming-based algorithms to optimize queries with expensive predicates.

The techniques discussed above deal with UDF optimization when the functions appear in the *where* clause of an SQL query. Expensive UDFs can also appear in the projection clause as operations on ADTs. Let us consider the following example taken from [95]. A table stores images in a column and offers functions to crop an image and apply filters to it (e.g., a sharpening filter). Consider the following query (using the syntax syntax of [95]):

```
select image.sharpen().crop(0.0, 0.0, 0.1, 0.1) from image_table
```

The user requests that all images be filtered using *sharpen*, and then *cropped* in order to extract the portion of the image inside the rectangular region with diagonal end-points (0.0, 0.0) and (0.1, 0.1). Sharpening an image first and then cropping is wasteful as the entire image would be sharpened and 99% of it would be discarded later (assuming the width and height of the images are 1.0). Inverting the execution order to *image.crop(0.1,0.1).sharpen()* reduces the total CPU cost. It may not be always possible for the user to enter the operations in the best order (i.e., *crop* function before the *sharpen* function), specially in the presence of relational views defined over the underlying tables [95]. In such cases, the optimizer should reorder these functions to optimize the CPU cost of the query. The Predator project [96] proposes to use Enhanced-ADTs or E-ADTs to address the above problem. The E-ADTs provide information regarding the execution cost of various operations, their properties (i.e., commutativity between *sharpen* and *crop* operations in the above example), etc., which the optimizer can use to reorder the operations and reduce the execution cost of the query. In some cases, it might be possible to remove function calls. For example, if X is an image, *rotate()* is a function that rotates an image and *count_different_colors()* is a function that counts the number of different colors in an image, the operation *X.rotate().count_different_colors()* can be replaced by *X.count_different_colors()*, thus saving the cost of rotation. [95] documents performance improvements of up to several orders of magnitude using these techniques.

5.3.2 Query Processing

In the previous section, we discussed how pushing the top-k query support into the database engine can lead to choice of better query evaluation plans. In this section, we discuss algorithms that the query processor can use to execute top-k queries for some of the query evaluation plans discussed in the previous section.

Evaluating Top-k Queries Let us again consider an image database with two features, color and texture, each of which is indexed using a multidimensional index structure. Let F_{color} and $F_{texture}$ denote the similarity functions for the color and texture features individually. Examples of individual feature similarity functions (or equivalent distance functions) are the various L_p metrics. Let F_{agg} denote the function that combines (or aggregates) the individual similarities (with respect to color and texture features) of an image to the query image to obtain its overall similarity to the query image. Examples of aggregation functions are weighted summation, probabilistic and fuzzy conjunctive and disjunctive models etc. [82]. The functions F_{color}, $F_{texture}$ and F_{agg} and their associated weights together constitute the retrieval model (cf. Section 2.3). In order to support top-k queries inside the database engine, the engine must allow users to plug in their desired retrieval models for the queries and tune the weights and the functions in the model at query time. The query optimization and evaluation must be based on the retrieval model specified for the query. We next discuss some query evaluation algorithms that have been proposed for the various retrieval models.

One of the evaluation plans for this example database, discussed in Section 5.3.1, is to use the individual feature indexes to find the best matches with respect to each feature individually and then merge the individual results. Fagin [31] proposed an algorithm to evaluate a top-k query efficiently according to the above plan. The input to this algorithm is a set of ranked lists X_i generated by the individual feature indices (by the k-NN algorithm). The algorithm accesses each X_i in sorted order based on its score and maintains a set $L = \cap_i X_i$ that contains the intersection of the objects retrieved from the input ranked lists. Once L contains k objects, the full tuples of all the items in $\cup_i X_i$ are probed (by accessing the relation), their overall scores are computed and the tuples with the k best overall scores are returned. The above algorithm works as long as F_{agg} is monotonic i.e. $F_{agg}(x_1, \ldots, x_m) \leq F_{agg}(x'_1, \ldots, x'_m)$ for $x_i \leq x'_i$ for every i. Most of the interesting aggregation functions like the weighted sum model and the fuzzy and probabilistic conjunctive and disjunctive models satisfy the monotonicity property. Fagin also proposes optimizations to his algorithm for specific types of scoring functions [31].

While Fagin proposes a general algorithm for all monotonic aggregation functions, Ortega et al. [82, 81] propose evaluation algorithms that are tailored to specific aggregation functions (i.e., separate algorithms for weighted sum, fuzzy conjunctive and disjunctive models and probabilistic conjunctive and disjunctive models). This approach allows incorporating function-specific optimizations in the evaluation algorithms, and is used by the MARS [52] system. One of the main advantages of these algorithms is that they do not need to probe the relation for the full tuples. This can lead to significant performance improvements since, according to the cost model proposed in [31], the total database access cost due to probing can be much higher than the total cost due to sorted access (i.e. accesses using the individual feature indices). Another advantage comes from the *demand-driven data flow* followed in [82]. While Fagin's approach retrieves objects from the individual streams and buffers them until it reaches the termination condition ($|L| \geq k$) and then returns all the k objects to the user, the algorithms in [82] retrieve only the necessary number of objects from each stream in order to return the next best match. This demand-driven approach reduces the wait time of intermediate answers in temporary files or buffers between the operators in a query tree. On the other hand, in [31], the items returned by the individual feature indexes must wait in a temporary file or buffer until the completion of the probing and sorting process. Note that both approaches are incremental in nature and can support the "get more" feature efficiently. Several other optimizations of the above algorithms have been proposed recently [73, 43].

An alternative approach to evaluating top-k queries has been proposed by Chaudhuri and Gravano [24, 25]. It uses the results in [31] to convert top-k queries to *alpha-cut* queries and processes them as filter conditions. Under certain conditions (uniquely graded repository), this approach is expected to access no more objects than the strategy in [31]. Much like the former approach, this approach also requires temporary storage and sorting of intermediate answers before returning the results. Unlike the former approaches, this approach cannot support the "get more" feature without re-executing the query from scratch. Another way to convert top-k queries to threshold queries is presented in [29]. This work mainly deals with traditional datatypes. It uses selectivity information from the DBMS to probabilistically estimate the value of the threshold that would retrieve the desired number of items and then uses this threshold to execute a normal range query. Carey and Kossman propose techniques to limit the answers in an ORDER BY query to a user-specified number by placing stop operators in the query tree [14, 15].

Similarity Joins Certain queries in image retrieval applications may require join operations. An example is finding all pairs of images that are most similar to each other. Such joins are called similarity joins. A join between two relations returns every pair of tuples from the two relations that satisfy a certain selection predicate.[18] In a traditional relation join, the selection predicate is precise (i.e., equality between two numbers): hence, a pair of tuples either does or does not satisfy the predicate. This is not true for general similarity joins where the selection predicate is imprecise (i.e., similarity between two images): each pair of tuples satisfies the predicate to a certain degree but some pairs satisfy the predicate more than other pairs. Thus, just using similarity as the join predicate would return almost all pairs of images including pairs with very low similarity scores.

Since the user is typically interested in the top few best matching pairs, we must restrict the result to only those pairs with good matching scores. This notion of restricted similarity joins have been studied in the context of geographic proximity distance [5, 50, 4] and in the similarity context [12, 97, 2, 98, 85, 64]. The problem of a restricted similarity join between two datasets A and B containing d-dimensional points is defined as follows [64, 97]. Given points $X = (x_1, x_2, ..., x_d) \in A$ and $Y = (y_1, y_2, ..., y_d) \in B$ and a threshold distance ϵ, the result of the join contains all pairs (X, Y) whose distance $D_d^\epsilon(X, Y)$ is less than ϵ. The distance function $D_d^\epsilon(X, Y)$ is is typically an L_p metric (Chapter 14), but other distance measures are also possible.

A number of non-indexed and indexed methods have been proposed for similarity joins. Among the non-indexed ones, the nested loop join is the simplest but has the highest execution cost. If $|A|$ and $|B|$ are the cardinality of the datasets A and B then the nested loop join has a time complexity of $|A| \times |B|$, which degenerates to a quadratic cost if the datasets are similarity sized. Among the indexed methods, one of the earliest proposed ones is the R-Tree based method is presented in [12]. This R-Tree method traverses synchronously the structure of two R-Trees matching corresponding branches in the two trees. It expands each bounding rectangle by $\epsilon/2$ on each side along each dimension, and recursively searches each pair of overlapping bounding rectangles. Most other index-based techniques for similarity joins employ variations of this technique [97]. The generalization of the multidimensional similarity join technique described above (which can be applied to obtain similar pairs with respect to individual features) to obtain overall similar pairs of images remains a research challenge.

[18]Note that the two input relation to a join can be the same relation (known as a self join).

6 Conclusions

In this chapter, we explored how the evolution of traditional relational databases into powerful extensible object-relational systems has facilitated the development of applications that require storage of multimedia objects and retrieval based on their content. In order to support content-based retrieval, the representation of the multimedia object (object model) must capture its visual properties. A user formulates a content-based query by providing examples of objects similar to the ones he/she wishes to retrieve. Such a query is internally mapped to a feature-based representation (query model). Retrieval is performed by computing similarity between the multimedia object and the query based on the feature values, and the results are ranked by the computed similarity values. The key technologies provided by modern databases that facilitate the implementation of applications requiring content-based retrieval are the support for user-defined data types, including user-defined functions, and ways to call these functions from within SQL. Content-based retrieval models can be incorporated within databases using these extensibility options: the internal structure and content of multimedia objects can be represented in DBMSs as abstract data types, and similarity models can be implemented as user-defined functions. Most major DBMSs now support multimedia extensions (either developed in house or by third-party developers) that consist of predefined multimedia data types, and commonly used functions on those types including functions that support similarity retrieval. Examples of such extensions include the Image Datablade supported by the Informix Universal Server, and the QBIC extender of DB2.

While existing commercial object-relational databases have come a long way in providing support for multimedia applications, we believe that there are still many technical challenges that need to be addressed in incorporating multimedia objects into DBMSs. Among the primary challenges is that of extensible query processing and query optimization. Multimedia similarity queries differ significantly from traditional database queries — they deal with high dimensional data sets, for which existing indexing mechanisms are not sufficient. Furthermore, a user is typically interested in retrieving the top-k matching answers (possibly progressively) to the query. Many novel index methods that provide efficient retrieval over high-dimensional multimedia feature spaces have been proposed in the literature. Furthermore, efficient query processing algorithms for evaluating top-k queries have been developed. These indexing mechanisms and query processing algorithms need to be incorporated into database systems. To this end, database systems have begun to support extensibility options for users to add type-specific access methods. However, current mechanisms are limited in scope and quite cumbersome to use. For example, to incorporate a new access method, a user has to address the daunting task of concurrency control and recovery for the access method. Query optimizers are still not sufficiently extensible to support optimizing access path selection based on user-defined functions and access methods. Research on these issues is ongoing and we believe that solutions will be incorporated into future DBMSs, resulting in systems that efficiently support content-based queries on multimedia types.

7 Acknowledgements

This work was supported by NSF CAREER award IIS-9734300, and in part by the Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0003.

References

[1] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *Proc. 1999 ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, 1999.

[2] K. Alsabti, S. Ranka, and V. Singh. An efficient parallel algorithm for high dimensional similarity join. In *Proc. Int. Parallel and Distributed Processing Symp.*, Orlando, Florida, March 1998.

[3] P. Aoki. Generalizing "search" in generalized search trees. In *Proc. 14th Int. Conf. on Data Engineering*, pages 380–389, 1998.

[4] Walid G. Aref and Hanan Samet. Optimization for spatial query processing. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proc. 17th Int. Conf. on Very Large Data Bases VLDB '92*, pages 81–90, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.

[5] Walid G. Aref and Hanan Samet. Cascaded spatial join algorithms with spatially sorted output. In *Proc. 4th ACM Workshop on Advances in Geographic Information Systems*, pages 17–24, 1997.

[6] Jeffrey R. Bach, Charles Fuller, Amarnath Gupta, Arun Hampapur, Bradley Horowitz, Rich Humphrey, Ramesh Jain, and Chiao fe Shu. The virage image search engine: An open framework for image management. In *Proc. SPIE*, volume 2670, *Storage and Retrieval for Still Image Video Databases*, pages 76–87, 1996.

[7] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. *Building an Object-Oriented Database System: The Story of O2.* The Morgan Kaufmann Series in Data Management Systems, May 1992.

[8] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proc. 21st Int. Conf. on Very Large Data Bases VLDB '95*, pages 299–310, 1995.

[9] S. Berchtold, C. Bohm, D. Keim, and H. P. Kriegel. A cost model for nearest neighbor search in high dimensional data spaces. In *Proc. 16th ACM Symp. Principles of Database Systems, PODS '97*, pages 78–86, Tucson, AZ, May 1997.

[10] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proc. Int. Conf. Database Theory (ICDT '99)*, pages 217–235, Jerusalem, Israel, 1999.

[11] R. Bliuhute, S. Saltenis, G. Slivinskas, and C. Jensen. Developing a datablade for a new index. In *Proc. 15th Int. Conf. on Data Engineering*, pages 314–323, 1999.

[12] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-Trees. In Peter Buneman and Sushil Jajodia, editors, *Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, Washington, DC, USA, May 26-28 1993. ACM Press.

[13] John L. Bruno, Jose Carlos Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE Int. Conf. Multimedia Computing and Systems*, pages 400–405, June 1999.

[14] Michael J. Carey and Donald Kossmann. On Saying "Enough Already!" in SQL. In *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 219–230, 1997.

[15] Michael J. Carey and Donald Kossmann. Reducing the Braking Distance of an SQL Query Engine. In *Proc. 24th Int. Conf. on Very Large Data Bases VLDB '98*, pages 158–169, 1998.

[16] Michal J. Carey, David J. DeWitt, Daniel Frank, M. Muralikrishna, Goetz Graefe, Joel E. Richardson, and Eugene J. Shekita. The Architecture of the EXODUS extensible DBMS. In *Proc. of the 1986 Int. Workshop on Object-Oriented database systems*, pages 52–65, 1986.

[17] Chad Carson, Megan Thomas, Serge Belongie, Joseph M. Hellerstein, and Jitendra Malik. Blobworld: A system for region-based image indexing and retrieval. In *Proc. of the 3rd Int. Conf. on Visual Information Systems*, pages 509–516, June 1999.

[18] Kaushik Chakrabarti and Sharad Mehrotra. Dynamic granular locking approach to phantom protection in R-trees. In *Proc. 14th Int. Conf. on Data Engineering*, pages 446–454, February 1998.

[19] Kaushik Chakrabarti and Sharad Mehrotra. Efficient concurrency control in multidimensional access methods. In *Proc. 1999 ACM SIGMOD Int. Conf. on Management of Data*, pages 25–36, May 1999.

[20] Kaushik Chakrabarti and Sharad Mehrotra. The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces. In *Proc. 15th Int. Conf. on Data Engineering*, pages 440–447, March 1999.

[21] Kaushik Chakrabarti, Kriengkrai Porkaew, Michael Ortega, and Sharad Mehrotra. Evaluating Refined Queries in Top-k Retrieval Systems. *Available as Technical Report TR-MARS-00-04, University of California at Irvine, online at http://www-db.ics.uci.edu/pages/publications/*, July 2000.

[22] Donald D. Chamberlin. *A Complete Guide to DB2 Universal Database.* Morgan Kaufmann, July 1998.

[23] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. SEQUEL 2:A Unified Approach to Data Definition, Manipulation and Control. *IBM J. Research and Development*, 20(6):560–575, 1976.

[24] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. In *Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data*, pages 91–102, 1996.

[25] Surajit Chaudhuri and Luis Gravano. Evaluating top-k selection queries. In *Proc. 25th Int. Conf. on Very Large Data Bases VLDB '99*, pages 397–410, Edinburgh, Scotland, 1999.

[26] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. In *Proc. 22nd Int. Conf. on Very Large Data Bases VLDB '96*, pages 87–98, 1996.

[27] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[28] Stefan Dessloch and Nelson Mattos. Integrating SQL databases with content–specific search engines. In *Proc. 23rd Int. Conf. on Very Large Data Bases VLDB '97*, pages 528–537, Athens, Greece, 1997. IBM Database Technology Institute, Santa Teresa Lab.

[29] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top-n queries. In *Proc. 25th Int. Conf. on Very Large Data Bases VLDB '99*, pages 411–422, Edinburgh, Scotland, 1999.

[30] W. Niblack et. al. The QBIC project: Querying images by content using color, texture and shape. In *IBM Research Report*, February 1993.

[31] Ronald Fagin. Combining fuzzy information from multiple systems. In *Proc. 15th ACM Symp. Principles of Database Systems, PODS '96*, pages 216–226, 1996.

[32] Ronald Fagin and Edward L. Wimmers. Incorporating user preferences in multimedia queries. In *Proc. Int. Conf. Database Theory (ICDT '97)*, pages 247–261, 1997.

[33] C. Faloutsos, M. Flocker, W. Niblack, D. Petkovic, W. Equitz, and R. Barber. Efficient and effective querying by image content. Technical Report RJ 9453 (83074), IBM Research Report, Aug. 1993.

[34] Christos Faloutsos and Ibrahim Kamel. Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. In *Proc. 13th ACM Symp. Principles of Database Systems, PODS '94*, pages 4–13, 1994.

[35] Christos Faloutsos and Douglas Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, Dept. of Computer Science, Univ. of Maryland, 1995.

[36] Christos Faloutsos, Bernhard Seeger, Agma Traina, and Caetano Traina Jr. Spatial join selectivity using power laws. In *Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data*, pages 177–188, 2000.

[37] S. Flank, P. Martin, A. Balogh, and J. Rothey. Photofile: A Digital Library for Image Retrieval. In *Proc. 2nd Int. Conf. on Multimedia Computing and Systems*, pages 292–295, 1995.

[38] M. Flickner, Harpreet Sawhney, Wayne Niblack, and Jonathan Ashley. Query by Image and Video Content: The QBIC System. *IEEE Computer*, 28(9):23–32, September 1995.

[39] You-Chin (Gene) Fuh, Stefan Dessloch, Weidong Chen, Nelson Mattos, Brian Tran, Bruce Lindsay, Linda DeMichiel, Serge Rielau, and Danko Mannhaupt. Implementation of SQL3 Sturctured Type with Inheritance and Value Substitutability. In *Proc. 25th Int. Conf. on Very Large Data Bases VLDB '99*, pages 565–574, Edinburgh, Scotland, 1999.

[40] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation.* Prentice Hall, 1999.

[41] Vibby Gottemukkala, Anant Jhingran, and Sriram Padmanabhan. Interfacing parallel applications and parallel databases. In Alex Gray and Per-Åke Larson, editors, *Proc. 13th Int. Conf. on Data Engineering*, pages 355–364, Birmingham U.K, April 7-11 1997. IEEE Computer Society.

[42] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[43] U. Guntzer, W. Balke, and W. Kiebling. Optimizing Multi-Feature Queries for Image Databases. In *Proc. 26th Int. Conf. on Very Large Data Bases VLDB 2000*, pages 419–428, 2000.

[44] Amarnath Gupta and Ramesh Jain. Visual Information Retrieval. *Communications of the ACM*, 40(5):70–79, 1997.

[45] A. Guttman. R-tree: a dynamic index structure for spatial searching. In *Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[46] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 377–388, 1989.

[47] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees in database systems. In *Proc. 21st Int. Conf. on Very Large Data Bases VLDB '95*, pages 562–573, September 1995.

[48] Joseph M. Hellerstein. Practical predicate placement. In *Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data*, pages 325–335, 1994.

[49] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 267–276, 1993.

[50] G. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 237–248, Seattle, WA, USA, June 1998.

[51] M.K̃. Hu. *Visual Pattern Recognition by Moment Invariants, Computer Methods in Image Analysis.* IEEE Computer Society.

[52] Thomas S. Huang, Sharad Mehrotra, and Kannan Ramchandran. Multimedia analysis and retrieval system (MARS) project. In *Proc of 33rd Annual Clinic on Library Application of Data Processing - Digital Image Access and Retrieval*, 1996.

[53] Informix. *Getting Started with Informix Universal Server, Version 9.1.* Informix, March 1997.

[54] Informix. *Informix Universal Server – Datablade Programmer's Manual Version 9.1.* Informix, March 1997.

[55] Informix. *Excalibur Image DataBlade Module User's Guide, Version 1.2.* Informix, 1999.

[56] Informix. *Media360, Any kind of content. Everywhere you need it.* Informix, 2000.

[57] Yoshiharu Ishikawa, Ravishankar Subramanya, and Christos Faloutsos. Mindreader: Querying databases through multiple examples. In *Proc. 24th Int. Conf. on Very Large Data Bases VLDB '98*, pages 218–227, 1998.

[58] H. Jiang, D. Montesi, and A. Elmagarmid. Videotext database system. In *IEEE Proc. Int. Conf. on Multimedia Computing and Systems*, pages 344–351, June 1997.

[59] Setrag Khoshafian and A.B. Baker. *Multimedia and Imaging Databases.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[60] Won Kim. Unisql/x unified relational and object-oriented database system. In *Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data*, page 481, 1994.

[61] Robert R. Korfhage. *Information Storage and Retrieval.* Wiley Computer Publishing, 1997.

[62] Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, Eliot Siegel, and Zenon Protopapas. Fast nearest neighbor search in medical image databases. In *Proc. 22nd Int. Conf. on Very Large Data Bases VLDB '96*, pages 215–226, 1996.

[63] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in generalized search trees. In *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 62–72, 1997.

[64] Nick Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *Proc. 14th Int. Conf. on Data Engineering*, pages 466–475, 1998.

[65] Gerald Kowalski. *Information Retrieval Systems: theory and implementation.* Kluwer Academic Publishers, 1997.

[66] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Communications of the ACM*, 34(63):50–63, October 1991.

[67] Chung-Sheng Li, Yuan-Chi Chang, John R. Smith, Lawrence D. Bergman, and Vittorio Castelli. Framework for efficient processing of content-based fuzzy cartesian queries. In *Proc. SPIE*, volume 3972, *Storage and Retrieval for Media Databases 2000*, pages 64–75, 2000.

[68] David B. Lomet. Key range locking strategies for improved concurrency. In *Proc. 19th Int. Conf. on Very Large Data Bases VLDB '93*, pages 655–664, August 1993.

[69] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 448–459, 1998.

[70] C. Mohan. ARIES/KVL: A key value locking method for concurrency control of multiaction transactions operating on B-tree indexes . In *Proc. 16th Int. Conf. on Very Large Data Bases VLDB '92*, pages 392–405, August 1990.

[71] Apostol Natsev, Rajeev Rastogi, and Kyuseok Shim. Walrus: A similarity retrieval algorithm for image databases. In *Proc. 1999 ACM SIGMOD Int. Conf. on Management of Data*, pages 395–406, 1999.

[72] J. Neivergelt et al. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Systems (TODS)*, 9(1):38–71, March 1984.

[73] Surya Nepal and M.V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proc. 15th Int. Conf. on Data Engineering*, pages 22–29, 1999.

[74] ORACLE. *Oracle 8i Concepts, Release 8.1.6.* Oracle, December 1999.

[75] ORACLE. *Oracle 8i* inter*Media Audio, Image and Video, User's Guide and Reference.* Oracle, February 1999.

[76] ORACLE. *Oracle 8i Visual Information Retrieval, Users Guide and Reference.* Oracle, 1999.

[77] ORACLE. *Oracle Data Cartridge, Developers's Guide, Release 8.1.5.* Oracle, February 1999.

[78] ORACLE. *Oracle Internet File System, Developers's Guide, Release 1.0.* Oracle, May 2000.

[79] ORACLE. *Oracle Internet File System, User's Guide, Release 1.0.* Oracle, April 2000.

[80] J. Orenstein and T. Merett. A class of data structures for associative searching. In *Proc. 3rd ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pages 181–190, 1984.

[81] Michael Ortega, Yong Rui, Kaushik Chakrabarti, Sharad Mehrotra, and Thomas S. Huang. Supporting Similarity Queries in MARS. In *Proc. of ACM Multimedia 1997*, pages 403–413, 1997.

[82] Michael Ortega, Yong Rui, Kaushik Chakrabarti, Kriengkrai Porkaew, Sharad Mehrotra, and Thomas S. Huang. Supporting Ranked Boolean Similarity Queries in MARS. *IEEE Trans. Knowledge and Data Engineering*, 10(6):905–925, December 1998.

[83] Michael Ortega-Binderberger, Sharad Mehrotra, Kaushik Chakrabarti, and Kriengkrai Porkaew. WebMARS: A Multimedia Search Engine for Full Document Retrieval and Cross Media Browsing. In *6th International Workshop on Multimedia Information Systems (MIS'00)*, pages 72–81, October 2000.

[84] Bernd-Uwe Pagel, Flip Korn, and Christos Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. 16th Int. Conf. on Data Engineering*, pages 589–598, 2000.

[85] Apostolos N. Papadopoulos and Yannis Manolopoulos. Similarity query processing using disk arrays. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 225–236, 1998.

[86] A. Pentland, R.W. Picard, and S.Sclaroff. Photobook: Content-based manipulation of image databases. *International Journal of Computer Vision*, 18(3):233–254, 1996.

[87] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. 23rd Int. Conf. on Very Large Data Bases VLDB '97*, pages 486–495, 1997.

[88] Kriengkrai Porkaew, Kaushik Chakrabarti, and Sharad Mehrotra. Query refinement for content-based multimedia retrieval in MARS. *Proc. of the 7th. ACM Int. Conf. on Multimedia*, pages 235–238, 1999.

[89] Kriengkrai Porkaew, Sharad Mehrotra, Michael Ortega, and Kaushik Chakrabarti. Similarity search using multiple examples in MARS. In *Proc. Int. Conf. on Visual Information Systems*, pages 68–75, 1999.

[90] F. Rabitti and P. Stanchev. GRIM DBMS: A GRaphical IMage Database Management System. In *Visual Database Systems, IFIP TC2/WG2.6 Working Conference on Visual Database Systems*, pages 415–430, 1989.

[91] Yong Rui, Thomas S. Huang, Michael Ortega, and Sharad Mehrotra. Relevance feedback: A power tool for interactive content-based image retrieval. *IEEE Trans. Circuits and Systems for Video Technology*, 8(5):644–655, September 1998.

[92] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1983.

[93] Thomas Seidl and Hans-Peter Kriegel. Optimal multistep k-nearest neighbor search. In *Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, pages 154–165, 1998.

[94] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, Boston, MA, USA, May 30 - June 1 1979. ACM.

[95] Praveen Seshadri. E-ADTs and Relational Query Optimization. Technical report, Cornell University Technical Report, June 1998.

[96] Praveen Seshadri. Enhanced abstract data types in object-relational databases. *VLDB Journal*, 7(3):130–140, August 1998.

[97] John C. Shafer and Rakesh Agrawal. Parallel algorithms for high-dimensional proximity joins for data mining applications. In *Proc. 23rd Int. Conf. on Very Large Data Bases VLDB '97*, pages 176–185, 1997.

[98] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. High-dimensional similarity joins. In *Proc. 13th Int. Conf. on Data Engineering*, pages 301–311, 1997.

[99] John R. Smith and S.-F. Chang. Integrated spatial and feature image query. *Multimedia Systems Journal*, 7(2):129–140, 1997.

[100] John R. Smith and S.-F. Chang. Visually Searching the Web for Content. *IEEE Multimedia*, 4(3):12–20, Summer 1997.

[101] T. G. Aguierre Smith. Parsing movies in context. In *Summer Usenix Conference, Nashville, Tennessee*, pages 157–167, 1991.

[102] J. Srinivasan, R. Murthy, S. Sundara, N. Agarwal, and S. DeFazio. Extensible indexing: A framework for integrating domain-specific indexing schemes into oracle8i. In *Proc. 16th Int. Conf. on Data Engineering*, pages 91–100, 2000.

[103] M. Stonebraker and G. Kemnitz. The postgres next-generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[104] Michael Stonebreaker and Dorothy Moore. *Object-Relational DBMSs, The Next Great Wave.* Morgan Kaufman, 1996.

[105] Michael J. Swain. Interactive indexing into image databases. In *1908,* Storage and Retrieval for Image and Video Databases, volume 1908, pages 95–103, 1993.

[106] Sybase. *Java in Adaptive Server Enteprise, Version 12.* Sybase, October 1999.

[107] Sybase. *Sybase Adaptive Server Enterprise, Version 12.0.* Sybase, 1999.

[108] Hideyuki Tamura et al. Texture features corresponding to visual perception. *IEEE Trans. Systems, Man and Cybernetics*, SMC-8(6):460–473, June 1978.

[109] Yannis Theodoridis and Timos Sellis. A Model for the Prediction of R-tree Performance. In *Proc. 15th ACM Symp. Principles of Database Systems, PODS '96*, pages 161–171, 1996.

[110] L. Wu, C. Faloutsos, K. Sycara, and T. Payne. Falcon: Feedback adaptive loop for content-based retrieval. In *Proc. 26th Int. Conf. on Very Large Data Bases VLDB 2000*, pages 297–306, 2000.