Bulletin of the Technical Committee on

# Data Engineering

December 1996 Vol. 19 No. 4

IEEE COMPUTER SOCIETY 50 YEARS OF SERVICE • 1946-1996

## Letters

Letter from the Editor-in-Chief . . . . . . . . *David Lomet* 1
Letter from the Special Issue Editor . . . . . . . . *Joseph M. Hellerstein* 2

## Special Issue on Query Processing for Non-Standard Data

Query Processing in a Parallel Object-Relational Database System . . . . . . . . *Michael A. Olson, Wei Michael Hong, Michael Ubell, Michael Stonebraker* 3
E-ADTs: Turbo-Charging Complex Data . . . . . . . . *Praveen Seshadri, Miron Livny, Raghu Ramakrishnan* 11
Storage and Retrieval of Feature Data for a Very Large Online Image Collection . . . . . . . . *Chad Carson and Virginia E. Ogle* 19
Data Modeling and Querying in the PIQ Image DBMS . . . . . . . . *Uri Shaft and Raghu Ramakrishnan* 28
An Optimizer for Heterogeneous Systems with NonStandard Data and Search Capabilities . . . . . . . . *Laura M. Haas, Donald Kossman, Edward L. Wimmers, Jun Yang* 36
Optimizing Queries over Multimedia Repositories . . . . . . . . *Surajit Chaudhuri and Luis Gravano* 44

## Conference and Journal Notices

International Conference on Data Engineering . . . . . . . . back cover

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

# Letter from the Editor-in-Chief

## Bulletin Related Items

### Membership Renewal

I want to re-emphasize that the Technical Committee Membership Application Form (printed with the September 1996 issue) must be filled in and forwarded to the Computer Society in order for you to continue as a member of the TC on Data Engineering in 1997. It is essential that you file this form in order for you to continue to receive the Data Engineering Bulletin. You may also renew by visiting the IEEE Computer Society web site at

*http://www.computer.org/tab/tcapplic.htm*

and filling in the application there.

Below is the message that I received from the Computer Society.

```
MEMBERSHIP RENEWAL BY DECEMBER 31, 1996 -- To serve our membership the
best way we can with limited resources, we must periodically ensure
that those of you who receive this newsletter and other information
continue to be interested in being part of TCDE.  Every few years, we
will ask that you renew your membership in the technical committee to
remain on the list.

Renew now by completing and returning the TC application form
by mail, fax, or email.  You only need to submit one form for
all TCs you belong to, as long as you check the appropriate boxes (up
to four).  Members who do not respond by December 31, 1996 will be
removed from the TC list(s).
```

### Editorial Appointment

I am happy to once again announce the appointment of another Associate Editor. The new editor is Daniel Barbara from Bellcore. Daniel has an outstanding research record over a number of years, and was in the very productive MITL group working on the handling of new types of data. Daniel recently moved to Bellcore. His current interests include both new types of data and on-line analytic processing (OLAP). Daniel will be the editor of the March 1997 issue of the Bulletin.

## About this Issue

The current issue focuses on query processing that involves new kinds of data. I am sure that reading the articles will convince you that query processing continues to be an exciting research area. This is true not only in terms of what has been accomplished so far, but also in how much remains to be done. Indeed, there is an open-endedness to the agenda as new forms of data will continue to emerge. Joe Hellerstein has brought together a fine collection of articles from leading researchers that provides us with a snapshot of the current state of query processing art for new data. Joe deserves some extra thanks for having to deal with the additional problem of handling image data in an issue of the Bulletin- which had its challenging moments.

David Lomet
Microsoft Corporation

## Letter from the Special Issue Editor

In the first part of this decade, the database community completed an initial phase of research into systems for non-standard data. This first phase — epitomized by prototype systems like Postgres, Starburst, and Exodus — focused largely on *enabling* technologies, i.e. architectures and paradigms to make it possible for database systems to handle complex data types. These projects and others demonstrated that database systems could indeed be extended to store and retrieve diverse data for applications such as multimedia, CAD, biotechnology, and earth science. A key goal of this research was to support ad-hoc querying over non-standard data, and this was achieved with varying degrees of success in the initial prototypes.

This issue of IEEE Data Engineering Bulletin focuses on more recent efforts in processing queries over non-standard data. This ongoing second phase of research aims not only for more functionality than was offered in the first phase, but also for increased performance.

Our first two papers focus on broad architectural issues, but from very different viewpoints. The first paper is a representative description of the state of the art in industrial systems. Informix Universal Server is an Object-Relational DBMS (ORDBMS) with an impressive pedigree: it combines Informix's commercial shared-memory parallel RDBMS technology with the initial commercialization of the Postgres research system. In this paper, the authors consider a number of challenges to the integration of parallel and object-relational technology that were raised by DeWitt in a recent talk. The second paper by Seshadri, *et al.* presents a new architecture being developed in academia. Their approach for "Enhanced" ADTs revisits the initial design assumptions of Postgres and Starburst, proposing a new multi-layered, multi-model approach that allows for more aggressive optimization.

Our second pair of papers focus on a particular application: image databases. Carson and Ogle describe work done in the UC Berkeley Digital Library project, which uses a commercial ORDBMS for storage and content-based querying of image data. Berkeley's Digital Library project includes novel feature-extraction research from the AI community, and the interaction between that research and a commercial ORDBMS makes for some interesting tradeoffs. Shaft and Ramakrishnan describe a data modeling approach to image storage and query processing, in which much of the semantic work is done during schema design. This is particularly attractive in that it circumvents some of the limitations of AI techniques for feature extraction, by leveraging the "natural" intelligence of the DBA in a relatively convenient and powerful fashion.

Our final pair of papers focus on one of the most challenging areas in handling queries over complex data: query optimization. Haas, et al. describe techiques used in the Garlic system to optimize queries over heterogenous data repositories, based on simple descriptions of the component systems. In addition to its focus on query optimization, this paper presents a third architectural alternative to be compared with those of our first two papers. Chaudhuri and Gravano describe techniques for optimizing queries over multimedia databases. Their focus is to integrate the optimization of expensive content-based predicates with the type of result ranking that is traditionally done in information retrieval systems. Again, this brings ideas from the AI domain and recasts them within a database framework; in this case the benefit is to introduce new functionality to databases, and also to realize increased optimization opportunities.

This is an exciting time for this branch of database research. The basic enabling technology is fairly mature, and is being implemented seriously in industrial settings. With that research complete, a host of issues have arisen — both efficiency challenges left over from the original designs, and problems of incorporating ever more functionality to leverage the semantics of complex data. The articles collected here form an interesting introduction to this growing body of work.

Joseph M. Hellerstein
U.C. Berkeley

# Query Processing in a Parallel Object-Relational Database System

Michael A. Olson   Wei Michael Hong   Michael Ubell   Michael Stonebraker
Informix Software, Inc.

**Abstract**

*Object-relational database systems are now being deployed for real use by customers. Researchers and industry users have begun to explore the performance issues that these systems raise. In this paper, we examine some of those performance issues, and evaluate them for object-relational systems in general and for INFORMIX-Universal Server in particular. We describe object-relational query processing techniques and make some predictions about how they will evolve over the next several years.*

## 1 Introduction

Since the middle 1970s, with the advent of the relational model and the appearance of working systems with which to experiment, academic and industry researchers have spent considerable time and energy inventing ways to make queries in relational databases run faster. Most relational database systems were sold to businesses, so most researchers concentrated on speeding up business query processing. The creation and use of indices, access path selection, cache structure and management, and the introduction of parallelism have all, over the years, attracted the attention of researchers. Innovations in these areas and others have produced a wide variety of fast, inexpensive, and powerful relational database systems.

Beginning roughly in 1985, with work on POSTGRES [Ston86] and Starburst [Haas90], researchers began to investigate a different kind of performance. Commercial relational systems were designed to execute queries quickly. These new research efforts were, fundamentally, designed to adapt quickly to changing business requirements, such as the need to model evolving business rules and manage new kinds of data. Systems of this kind, which have since been dubbed *object-relational*, preserved the relational model on which they had been built, but added support for developers to define new data types and operations. Object-relational systems have maintained backward compatibility with conventional relational engines. They can be used anywhere that relational engines can, because they provide the same operations and abstractions as relational engines. Object-relational engines use SQL3 [ANSI96], a superset of the query language supported by relational engines.

At the same time, object-relational systems borrowed heavily from work on object-oriented programming languages and systems.

Early object-relational systems emphasized extensibility over query processing performance. Since the beginning of the 1990s, however, and particularly with the introduction of Illustra to the commercial marketplace,

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

object-relational systems have attracted the attention of the major database vendors and customers. Today, vendors are working hard to turn their relational offerings into object-relational products, and to make sure that those products are fast. In addition, industry and academic researchers are focusing their attention on query processing in object-relational systems.

In this paper, we draw on our industry and research experience to describe query processing strategies in object-relational systems, and to make some predictions about how such systems will evolve over the next several years. We summarize some of the major features of the INFORMIX-Universal Server, the first commercially-available parallel object-relational database system. We explain why object-relational systems are better than relational ones for high-performance query processing, and in particular how object-relational systems can take advantage of parallelism in interesting new ways.

## 2 Comparing Relational to Object-Relational

Because object-relational systems are still fairly new in the marketplace, discussion of them is often more vigorous than informed. In the sections that follow, we provide a brief comparison of relational and object-relational systems. This comparison provides a basis for evaluating both our claims and those of others on what object-relational systems can do.

A much more detailed comparison of the two architectures appears in [Ston96].

### 2.1 Object-Relational Query Processing

At base, object-relational systems use all the same query processing techniques as relational systems do. In particular, access path selection in object-relational systems works just like it does in relational systems. A cost-based optimizer considers a number of strategies for evaluating a query, and the least expensive plan is passed to the query execution engine. The executor supports the same scan and join strategies as do relational systems.

The most significant difference between relational and object-relational systems is that object-relational systems can always be extended with new code to store and operate on new data types. This extensibility is fundamental to object-relational data management, and permeates the entire engine. For example, an object-relational cost-based query optimizer must recognize the cost of operations provided by some extension, and must factor that cost into the total query cost.

Because object-relational systems are extensible, the number and variety of queries that they can execute is unbounded. Users may write queries over any data type, or combination of data types, known to the engine. As new data types are developed, they may be added to the system, and are available to users in the same way as those built into the system by the database system vendor.

A second important difference between object-relational and relational systems is that object-relational systems allow the definition of new primary and secondary storage structures. Developers may implement new ways of storing tables, and new index structures, to take advantage of special hardware, external software, or the properties of the data to be stored. A common example is the creation of spatial indexing structures to support fast searches on geographical or geometric data.

These new tables and indices give object-relational engines a potentially infinite number of ways of scanning user data. The query processing engine can choose an access path that uses a new index or table storage to execute a query. An object-relational query optimizer understands the performance characteristics of the new storage structures, and can use the same sort of cost metrics that relational systems do in order to produce a good query plan.

There are other ways in which object-relational engines improve on the query processing techniques afforded by strictly relational engines. Because they include all the components of a relational system, object-relational engines provide a strict superset of the query processing techniques available in relational engines.

## 2.2 Parallelism

All the major relational vendors provide support for *inter-query parallelism*. Inter-query parallelism allows multiple queries to execute in parallel, with some degree of resource sharing among them. For example, most vendors have a single shared buffer cache which all active queries use, and pages read into the cache by one query are available to any other query.

Some relational vendors also provide *intra-query parallelism*. Intra-query parallelism allows a single query to be split into multiple threads of execution, which can then run in parallel.

Because of their heritage, object-relational systems offer one or both of these varieties of parallelism. INFORMIX-Universal Server offers inter- and intra-query parallelism, as it is based on Informix's relational product, OnLine, and OnLine provides both kinds of parallelism.

Very few relational vendors offer *inter-function parallelism*. It allows a pipeline of functions in a query to be executed in parallel.

For example, consider an image processing pipeline, in which an image is clipped, scaled, and recolored to reduce the size of the palette it requires. A naive implementation of this process would be to clip the entire image, then scale the clipped version, then recolor the result. This strategy uses a lot of space, since at least two copies of the image must be in memory or on disk at each step. In addition, each function must wait for the one before it to complete before it can begin its work.

A better strategy is to use inter-function parallelism to change the way the pipeline is executed. Instead of operating on an entire image, each function in the pipeline operates on sets of scanlines. In this case, each function can be bound to a single thread of execution, and they can pass single scanlines among themselves. As a result, only a small amount of memory is required at each step, and the system can run the three operations in parallel.

Finally, no relational systems support *intra-function parallelism*. Intra-function parallelism allows a function over a single value to be broken into multiple threads of execution, so that each thread operates on part of the value at the same time.

Again, consider an image processing function. Scaling a large image up or down requires reading the entire image, but groups of scanlines can be scaled independently of one another. To scale an image down by a factor of two in both dimensions, for example, requires that each block of four by four pixels be replaced by a single pixel in the result. Computing the new pixel from the originals can proceed in parallel, since the result pixels are all independent of one another.

A well-designed object-relational system offers all these kinds of parallelism, because all are important to good performance. More importantly, the system must make all of them available to developers who write extensions. Inter-query parallelism is important because multiple users will want to use an image repository at the same time, just like multiple users want access to a payroll database managed by a relational engine. Intra-query parallelism is important to object-relational customers because object-relational databases can be very large, and only intra-query parallelism delivers good performance as the database grows. Intra- and inter-function parallelism are important because many popular extensions to object-relational systems support expensive functions on large data values.

## 3 Performance Issues in Parallel Object-Relational Systems

Object-relational engines allow developers, most of whom are not database server implementers, to extend the engine with new types and functions. Naturally, this power comes at some cost. Designers must consider the behavior and performance characteristics of each extension, and of combinations of these extensions, to guarantee that deployed systems will meet demands.

Query processing performance in object-relational systems has recently captured the attention of researchers. Work on characterizing performance has so far mostly used synthetic workloads, because object-relational systems have not been very widely deployed in commercial situations. As a result, the performance issues raised to date are interesting, but far from complete. Only with widespread production use of these systems will their real strengths and weaknesses become apparent. As a result, it is critical that object-relational engines continue to evolve, so that they can meet the demands of workloads that no one has yet foreseen.

With that caveat, [Dewi96] summarizes the most interesting performance questions confronting object-relational systems today. In the next several sections, we describe in detail how object-relational systems are able to overcome the problems cited in [Dewi96], and how they will change over the next several years as new problems arise.

## 3.1 Skew

*Data skew* is a well-known problem in relational systems. It arises when a system designer assumes that the distribution of data in a database system will follow one pattern, but the actual distribution follows a different pattern.

A common example of data skew is in the way that employees are assigned to departments in companies. Most technology companies, for example, employ many more engineers than accountants. As a result, the employee distribution is skewed toward engineering. If a database designer assumes that all departments will have the same number of employees, then query performance against this database may be bad.

Database systems that provide intra-query parallelism typically use data partitioning to provide distinct input streams for multiple threads of execution in a single query. This means that a single table may be spread across multiple disks, and that a single query can execute on each table fragment in parallel. The separate table fragments provide good performance because the system can read from multiple disks in parallel, which reduces aggregate I/O time, and because the threads can take advantage of multiple processors on the machine, if they are available.

Skew becomes a real problem in parallel systems when the distribution of data across disks is skewed with respect to queries. For example, if employees in our example are distributed across disks based on the department in which they work, then records for all engineers will be stored on one disk, and records for all accountants on another. Queries over all employees will not benefit very much from data partitioning, because the thread that examines engineering records will take much longer to complete than the thread that examines accounting records.

Relational database systems allow designers to partition data in three basic ways to handle this problem. Data may be partitioned in round-robin fashion, with new records being assigned to each disk in turn. It may be partitioned based on a hash value computed from the source key; this provides, effectively, random distribution, except that identical values are clustered on the same disk. Finally, data may be partitioned based on value ranges, so that employees who make between zero and twenty thousand dollars a year are assigned to one disk, with other salary ranges assigned to other disks.

[Dewi96] asserts that object-relational systems suffer more severely from skew than do relational systems. As an example, the paper draws on several novel features of object-relational systems, and describes how they can further skew distributions or lead to queries that do not match data distributions.

The INFORMIX-Universal Server uses data partitioning to provide intra-query parallelism and supports all the features that [Dewi96] uses in its example on skew. In order to understand the example, and why it is incorrect, it is important to understand the object-relational features that motivate it.

First, schema designers may create *row types*. Row types are records that may be stored and manipulated as atomic values in a column of a table. Thus a schema may include nested rows.

Second, a designer may use a *set* of values as a single value. Again, relational systems force designers to implement one-to-many foreign key relationships for a similar result. Object-relational systems make this easier to use by providing syntax and behavior that more closely matches the user's conceptual schema.

Using these features, [Dewi96] defines the following schema:

```
create row type emp_t (name varchar, salary money);
create table depts (name varchar, emps set(emp_t))
fragment by round robin in disk1, disk2, ...;
```

As in relational systems, if employees are not evenly distributed among departments, this schema is skewed. In fact, if we replace the set notation in this example with explicit join keys, and then fragment employees by department, we can produce an identical data distribution using a relational engine.

[Dewi96] claims that object-relational systems suffer because they cannot support fragmentation strategies that store sets on different disks from the rows that contain them. In fact, the behavior of any particular object-relational system is, in this case, a detail of its implementation, and not a fundamental feature of object-relational systems. Object-relational systems can support fragmentation strategies that distribute sets with, or separately from, the rows that contain them.

In addition, object-relational systems allow schema designers to use more than the simple hash, range, and round-robin fragmentation strategies offered by relational systems. Since an ORDBMS is extensible, designers may create arbitrary functions over their types that assign them to buckets or value ranges based on knowledge of the types' structures or the likely distribution of real data. Schema designers may do hash- or range-based partitioning on the result of a function, not just on an atomic value. The same statement applies to values in sets, or in any other SQL3 collection type.

The important point is that schema designers can use the declarative syntax with which they are comfortable to control the partitioning of both tables and collections. In general, that will be sufficient to create a schema that performs well in practice. In some cases, a designer may have special knowledge of the data domains or distribution that a database will store. In those cases, the schema designer can easily add new classification or hashing functions to the system, and can use those functions to control data placement.

Furthermore, users automatically get the performance improvements of a parallel system over new data types. The engine understands how to provide both query and function parallelism. Developers who write extensions need not do anything special to take advantage of it. Users who query the system via SQL need not do anything special to take advantage of it. The support is built in, and invoked automatically.

Note that skew, and in particular the kind of unequal data distribution produced by the declarations above, is not new to ORDBMSs. These problems have been noticed and studied carefully in parallel relational systems for years. An ORDBMS provides much more control to a schema designer to handle skew and do intelligent data distribution than does a conventional RDBMS.

## 3.2 Bad Schema Design

To alleviate the problems in the schema declared above, [Dewi96] proposes creating separate tables for employees and departments, and partitioning both of them in round-robin fashion across disks in the database.

Given that schema, [Dewi96] correctly demonstrates that some queries will perform badly. In particular, if employee records are not clustered with the department records that contain them, then a query that computes the average salary of employees by department turns into effectively random I/O across all disks, and one message per record among the threads participating in the query.

Naturally, performance is terrible.

However, this problem is not due to any object-relational features. Rather, it is an example in which poor schema design leads to very bad query performance. Any physical organization of data on disk must consider the most common user queries, and data should be clustered to optimize those queries. This requirement has long been acknowledged for relational systems, and applies just as strongly to object-relational ones.

The basic problem with [Dewi96] is that it partitions data to support a particular workload, and then evaluates the schema using another workload entirely. Any system can provide at most one clustering strategy for data without copies. An object-relational system outperforms a relational one, if the two systems have identical data partitioning and query plans, because the object-relational system allows designers to choose among several models, such as explicit joins versus references, for performance or representational reasons.

## 3.3 Large Values

Finally, [Dewi96] explores performance of queries that operate over large data values by using three examples. Those examples provide a good demonstration of the superiority of object-relational systems to conventional ones, so we will preserve them in this discussion.

### 3.3.1 Time Series Data

The first example is time series data, such as information from a stock ticker. At regular or irregular intervals, new information arrives on the ticker, and should be included in the appropriate large time series. The SQL DDL

```
create table stocks {name varchar, prices timeseries(money));
```

creates a table that captures all the ticker traffic for a given stock name in a single record. As new price information arrives over the ticker, it is added to the appropriate timeseries value in the table.

Given this schema, for example, a user could write a query that computes the average closing price of a particular stock. [Dewi96] makes the point that if values in a timeseries are not partitioned across disks, then this query will get no benefit from parallelism.

That claim is correct, but is not an indictment of object-relational systems. Just as sets may be partitioned independently of the records that contain them, so may other constructed values. A timeseries, an array, a set, and a collection are all examples of constructed values, and users may want to partition the values stored in any of these. By partitioning timeseries values across disks, schema developers can take advantage of intra-function parallelism. Relational systems, which do not even support the new collection type, cannot offer the same functionality or performance.

### 3.3.2 Video

The next example in [Dewi96] uses video to argue that object-relational systems will not scale. If videos must be stored inside the database system, then playback speeds are likely to be compromised (since no object-relational system is designed to provide constant-bit-rate services), and operations on single videos will be slow, since videos cannot be partitioned.

The first claim is certainly correct; a video management strategy that stores videos in binary large objects should not be used for playback. In fact, any video management system that stores videos inside the database is likely to collapse under the weight of the data it holds. The files are huge. It takes only a few tens of minutes of compressed video to fill a two-gigabyte disk completely.

Object-relational systems allow the database to refer to values stored externally, on video servers or other special purpose hardware. The ORDBMS can fetch the contents from the repository when it must operate on them. The database engine stays out of the data path for playback. Naturally, there are issues of referential integrity and access control to consider, but the important point is that an ORDBMS is flexible enough to support both local and remote management of large data values.

### 3.3.3 Images

The third, and final, example type is image data. In this case, [Dewi96] claims that queries operating over images will run slowly, because the system has no way to parallelize expensive operations on single images.

In fact, an ORDBMS offers a number of ways to exploit parallelism. INFORMIX-Universal Server uses the strategy from Volcano, discussed in [Grae90], to represent in the query plan the points at which parallelism is possible. A type or function designer only needs to create a special class of function, called an *iterator*, in order to allow intra-operator parallelism. The INFORMIX-Universal Server considers available system resources and global optimization in order to decide where, and how, to exploit parallelism.

The best way to parallelize a sequence of operations over large data values is to break the value into a set of smaller values, and to have the pipeline operate over the members. For example, rather than writing functions that take and return a single image, a programmer can write functions that take and return a set of pixel rows. In that case, a sequence of operations, like scaling a clipped, resampled image, can execute in parallel, and only a small amount of data must be in memory at any time.

Once again, an ORDBMS provides designers with a variety of options to control the performance of their queries. The engine itself imposes no particular restrictions on how values are represented, and is able to exploit inter-query, intra-query, inter-function, and intra-function parallelism.

## 4 Conclusions

Object-relational systems are poised to replace relational systems in large-scale deployment over the next several years. As a result, researchers have begun to investigate the performance characteristics and design trade-offs in such systems.

Our experience in building, shipping, and supporting industrial-strength ORDBMSs convinces us that they are superior to their relational predecessors. This superiority is based on no single feature, but rather on the extreme flexibility of object-relational engines. They have been designed to support rapid innovation and maintain good performance on relational workloads. As a result, it is simple to experiment with ways of improving performance whenever new bottlenecks are discovered.

Object-relational systems give developers the tools that they need to build systems that perform well. A type designer, for example, can choose to create types and functions that support intra-function parallelism by using collection types, and building higher-level structures like images on top of lower-level structures like scanlines.

Similarly, schema designers have much better control over physical layout and logical data model than they do in relational systems. Because collection values can be partitioned independently of rows that contain them, the designer can lay out data to balance load across disks and to provide excellent query response for the most commonly-asked questions.

We believe that the currently-shipping object-relational systems address the performance problems raised in the research community now. In addition, because of the attention that object-relational systems must pay to extensibility, they will continue to perform well as experience grows and our understanding of workloads matures. This is because the constituent parts of a relational engine, like the query parser, planner, executor, and storage management, are well-separated and independent in an object-relational system. As a result, changes to the internals of any single component are less likely to affect the others, making rapid modifications easier.

## References

[ANSI96] ANSI X3H2 Committee. *Database Language SQL - Part 2: SQL/Foundation*, Part 2. Committee Draft, July 1996.

[Dewi96] Dewitt, David. Object-Relational and Parallel: Like Oil and Water? *Proceedings of Object-Relational Summit*, Miller Freeman, San Francisco, 1996.

[Grae90] Graefe, G. Encapsulation of parallelism in the Volcano query processing system. *Proceedings of ACM SIGMOD Conference*, ACM, New York, 1990.

[Haas90] Haas, L., *et al*. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering* 2(1), March 1990.
[Ston86] Stonebraker, M., and Rowe, L. The Design of POSTGRES. *Proceedings of the 1986 ACM-SIGMOD Conference*, ACM, New York, 1986.
[Ston96] Stonebraker, M., and Moore, D. *Object-Relational Databases: The Next Great Wave*, Morgan Kaufman, San Francisco, 1996.

# E-ADTs: Turbo-Charging Complex Data

Praveen Seshadri
Computer Science Dept.
Cornell University, Ithaca, NY
*praveen@cs.cornell.edu*

Miron Livny
Computer Sciences Dept.
U.Wisconsin, Madison WI
*miron@cs.wisc.edu*

Raghu Ramakrishnan
Computer Sciences Dept.
U.Wisconsin, Madison WI
*raghu@cs.wisc.edu*

**Abstract**

*The next generation of database applications will be dominated by rich and complex data types. The ADT technology of today's object-relational database systems cannot provide adequate performance for such applications. The basic stumbling block is the lack of semantic knowledge provided to the database system about the ADT operations. We are developing novel "Enhanced ADT" (E-ADT) technology that overcomes this problem, while retaining the extensibility of the database system. In effect, this "turbo-charges" the complex data types, resulting in dramatically improved performance. E-ADT technology is being demonstrated on several kinds of multi-media data in the $\mathcal{PREDATOR}$ database system at Cornell.*

## 1 Introduction

We are witnessing an explosion in the volume and complexity of digital information that people want to access and analyze. Much of this data is "multi-media" like images, video, audio, and documents. Several other kinds of complex data are also important, including geographical objects, chemical and biological structures, mathematical entities like matrices and equations, and financial data like time-series. Database systems must efficiently support queries over such richly structured data; otherwise, they will fast become "roadkill on the information super-highway" [DeW95]. To meet this challenge, novel "Enhanced ADT" (E-ADT) technology is being developed in the $\mathcal{PREDATOR}$ database system at Cornell.

The existing support for complex data in object-relational database systems (OR-DBMSs) is based on Abstract Data Types (ADTs), which were adapted from programming language concepts [LZ74] to databases in the 1980s [SRG83, Sto86]. Prominent OR-DBMSs based on ADTs include Illustra [Ill94], Postgres [SRH90] and Paradise [DKL$^+$94]. The OR-DBMS maintains a table of ADTs and new ADTs may be added by a database developer or user. Each ADT provides a number of primitive operations for manipulating or querying values of the type. An ADT for images might provide operations *Rotate(I, Angle), Clip(I, Region)*, and *Overlay(I1, I2)*, which are composed to form query expressions like *Overlay(Rotate(I1, 90), Clip(I2,* $\{$*0, 0, 100, 200*$\}$*))* that can be embedded within an SQL query. This is the current state of the art, which we call the "ADT approach".

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

## 2 Motivation

Operations on complex ADTs can be expensive (for example, *Rotate(Image)*); indeed, the cost of ADT operations often dominates the overall execution cost of a query. Obviously, query processing and optimization should attempt to reduce the cost of ADT operations. There are two issues to consider: (a) What are possible optimizations on such queries? (b) How can these optimizations be systematically applied in a DBMS?

### 2.1 Possible Optimizations

Consider an OR-DBMS based on the ADT approach, and assume that an image ADT has been added. While image data is stored on disk in a compressed format, the ADT operations are implemented on an uncompressed main-memory image data structure. Consequently, the image argument of any operation is converted to its main-memory uncompressed form before an operation is invoked on it.

An earth scientist maintains a table of geographic data, each entry having a satellite photograph and several other columns. An SQL query asks for a rotated portion of each photograph of the arctic region. The cost of this query is dominated by the operations on the images.

```
SELECT Clip(Rotate(G.Photo, 90), {0, 0, 100, 200})
FROM  GeoData G
WHERE G.Region = ’arctic’
```

How is this query evaluated in current OR-DBMSs[1]? For every data entry corresponding to the arctic region, the Photo attribute is retrieved from disk and decompressed into a main-memory image. The *Rotate* operation is then applied to it, and the resulting image is written to a compressed disk-resident form. The *Clip* operation is then applied with the intermediate result image as its input. This input image is decompressed to a main-memory form, it is clipped to the desired dimensions, and the resulting image is written out to disk. One could improve this execution strategy as follows:

- It is unnecessary for *Rotate* to compress and write its result to disk. Instead, it could be passed directly in memory to *Clip*.
- *Rotate* is an expensive operation, whose cost depends on the size of the image being rotated. It would be cheaper to evaluate the equivalent expression *Rotate(Clip(G.Photo, {0, 0, 200, 100}), 90)*. By performing *Clip* early, there can be significant reductions in the cost of *Rotate*. Note that the arguments of *Clip* have changed as a result of its being applied before *Rotate*.
- If *Clip* is applied before *Rotate*, the entire image does not need to be retrieved from disk. Only the appropriate portion of the image is needed.

A combination of these strategies can lead to performance improvements of an order of magnitude. Essentially, an entire ADT expression needs to be treated *declaratively* and optimized. The textual representation of an expression should not specify an evaluation plan. Current OR-DBMSs based on the ADT approach violate this principle; they do not perform such optimizations. Consequently, their performance is poor.

### 2.2 Categories of Optimizations

Similar improvements should be applied to ADT expressions involving other complex data types like audio, time-series, matrices, etc. We can classify the optimizations into four broad categories:

- *Algorithmic:* Using different algorithms for each operation depending on the data characteristics. For example, the best algorithm to use for the *Multiply* operation on two matrices depends on the sizes of the matrices and the amount of memory available.

[1]Our presentation is of the execution of a “typical” OR-DBMS like Illustra. Systems like Paradise [DKL$^+$94] incorporate some of the optimizations we suggest.

- *Transformational:* Changing the order of operations. The motivating example shows how *Clip* can be applied before *Rotate*.
- *Constraint:* Pushing constraints through the expression. The constraints may involve selecting a portion of the data, specifying a certain data resolution, or requiring a particular physical property. Consider the expression *Clip(Smoothen(I), Region)* on images. The whole image does not need to be smoothened; only a region of it is needed.
- *Pipelining:* Pipelined execution of operations to avoid materializing intermediate results. This is crucial for large data types like audio and video. It may be impossible to fit an entire uncompressed audio (or video) object in memory. The only reasonable way to access the data is as a stream. When a sequence of operations has to be applied to such a large object (for example, *IncrTreble(DecrVolume(Audio))*), pipelining the operations is clearly better than generating intermediate results for each operation. This is similar to lazy evaluation [Jon87] in functional programming.

## 3 A Performance Demonstration

What is the effect of such optimizations? We answer this with an initial performance demonstration in $\mathcal{PREDATOR}$. We use a data set containing 74 images of cars, compressed using JPEG. A Cars relation is created, with a name assigned to each image. Each image column value contains an Id for the image and its bounding box information. The size of the compressed images ranges from 23K to 266K, with an average of 65K. This is a relatively small amount of data, but it demonstrates the issues involved. The average size of the uncompressed images in memory is 0.8MB representing more than a 10-fold increase from the size on disk. Standard JPEG libraries are used to perform compression and decompression. Experiments were run on a Sparc20 machine with 64MB of physical memory, of which 8 MB was used as a database buffer pool. The buffer space was sufficient to hold the compressed data, and each individual uncompressed image easily fit in memory. The system is CPU-bound in all these experiments, with the expensive image operations using most of the execution time.

**Experiment 1:** The first experiment examines the effects of the Pipelining optimizations. We noted in Section 2 that in the ADT approach, each operation reads (and decompresses) its input from a disk-based representation, and writes (and compresses) its output. We call this strategy DISK. An improvement is to pass intermediate results in their main-memory form. We call this strategy MEM. Finally, if there is a sequence of operations, we could pipeline their execution by establishing an image row iterator. We call this strategy PIPE. We expect PIPE to be significantly better than MEM only when the intermediate results are larger than main-memory; that is not the case for these images.

```
SELECT Height(Negative(C.picture))
FROM Cars C;
```

In the query above, *Negative()*, which inverts the pixel values, is a relatively cheap operation. For all three strategies, *C.picture* must be read and decompressed. The DISK strategy compresses and writes the result out, whereas the MEM strategy does not. The *Height* operation is applied so that the display time for the result does not distort the measurements. The height is obtained from the image bounding box, so the actual image is not required. We gradually vary the query by introducing additional invocations of the *Negative()* operation. For instance, *SELECT Height(Negative(Negative(C.picture)))* requires one additional compression and decompression using the DISK strategy. The results are shown in Figure 1. Along the X-axis is the number of *Negative()* operations in the SELECT clause. The Y-axis shows the execution time.

As the number of operations in the image expression increases, the effect of the compression and decompression at the image boundaries dominates. Consequently, the MEM strategy which avoids these unnecessary costs is significantly cheaper than DISK. As expected, the PIPE strategy does only marginally better than MEM, since the intermediate results fit easily in memory. The MEM strategy, which is easier to implement, has been used as the default for the remaining experiments.

**Experiment 2:** The second experiment examines the effects of Order and Constraint optimizations. The

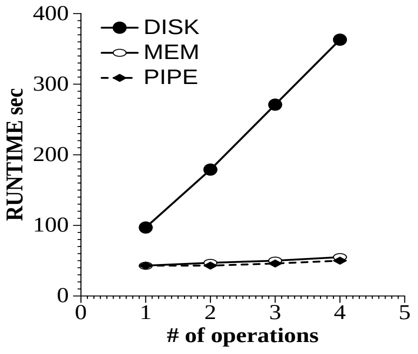


Figure 1: Pipelining

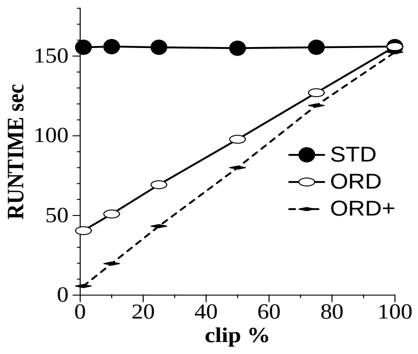


Figure 2: Varying Clip Size

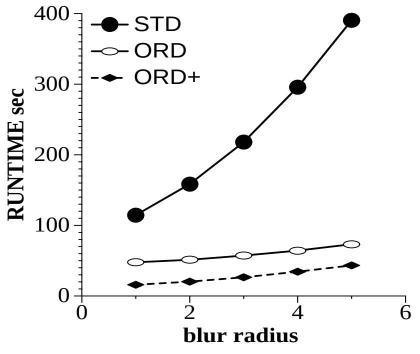


Figure 3: Varying Blur Radius

following query is used:

```
SELECT Height(Clip(Blur(C.picture, <radius>), <region>))
FROM Cars C;
```

The result of a *Blur* operation is an image in which every pixel's value is the average of the pixels within *radius* of it in the input image. *Blur* is an expensive operation, especially when *radius* is large. In the standard ADT approach, the images will first be blurred, then clipped to the appropriate region. We call this the STD strategy. Using the Order optimizations, the *Clip* operation could be performed before the *Blur* operation. We call this the ORD strategy. Finally, if *Clip* is being performed first, it can use the region to constrain the retrieval and decompression of the images. We call this the ORD+ strategy.

Figure 2 shows how these strategies perform when the *Blur radius* is fixed at 2 and the size of the Clip region is varied as a percentage of the image size. Since blurring an image is expensive, both ORD and ORD+ perform very much better than STD when a small region needs to be clipped. Further, it is evident that ORD+ is also significantly better than ORD because it requires a smaller portion of each image to be retrieved and decompressed. Figure 3 fixes the *Clip region* at 10% of the image size, and varies the *Blur radius* from 1 to 5. As the *radius* increases, the cost of the blurring grows quadratically. For the STD strategy, since this is the dominant cost, the overall execution time also grows quadratically. In comparison, in ORD and ORD+, the clips are performed early. The difference between ORD and ORD+ is the difference in the retrieval costs; it is evident that the execution time of ORD is dominated by the retrieval and decompression time. Consequently, the quadratic change in the *Blur* cost is not apparent.

**Experiment 3:** The third experiment demonstrates a very simple Constraint optimization so as to emphasize that constraints do not necessarily have to derive from a *Clip* operation. Consider the query below:

```
SELECT Area(Overlap(C1.picture, C2.picture))
FROM Cars C1, Cars C2
WHERE C2.name = "Alfa_Romeo_8C_2300"
```

The query needs to find the area of overlap between pairs of images. There is no need to compute the actual overlap of the images; their bounding boxes contain the necessary information to find the area of overlap. If this constraint is pushed into the computation of the *Overlap* operation, the performance is greatly improved. Measured execution time goes from 79.5 seconds to 0.58 seconds!

## 4 A Systematic Optimization Framework

Each of the demonstrated optimizations can result in an order of magnitude in performance improvements. Their combination can result in even greater improvements. We have shown similar improvements for queries involving time-series data [SLR96b]. However, it is not sufficient to observe that these optimizations are possible. There must also be an architectural framework to apply them in a correct and cost-based manner, so that the efficiency of a query is improved. At the same time, the type system should remain extensible, so that new types can be added incrementally. Continuing with the example using images, we note that:

- The best algorithms used for each operation can depend on the size of the input image, the amount of memory available, the storage format of the image and the values of arguments to the operation. Thus, for any complex expression involving multiple image operations, there are several possible evaluation plans. Deciding between plans requires cost-based optimization of the image expression.
- The image expression may be evaluated several times during the course of the query. Its evaluation plan should be chosen before the query starts executing, because it is unreasonable to explore the large space of options at runtime. Further, the cost of the chosen plan is used at compile time by the SQL optimizer to "place" the evaluation of the expression at the appropriate position in the join evaluation tree [Hel95, CS96].
- In order to have meaningful compile-time optimization, collective meta-information like the storage formats and size statistics need to be maintained over all the pertinent images (in this case, over all photographs in the GeoData table). The meta-information maintained and the optimizations to be applied are specific to each ADT.

Current OR-DBMSs lack such an optimization infrastructure. The E-ADT paradigm *does* provide the framework to support such optimizations. It "enhances" traditional ADTs so that combinations of ADT operations are treated as declarative expressions that are optimized in a cost-based manner. At the same time, the modularity of the type system is retained.

## 4.1 Enhanced Abstract Data Types

An Enhanced Abstract Data Type (E-ADT) enhances ADTs in database systems to improve the performance of query processing. In addition to standard ADT functionality, each E-ADT may support one or more of the following enhancements through a uniform internal interface:

*Query Language:* An E-ADT can provide parsing capabilities for a native query language with which expressions over values of that E-ADT can be specified. As an important special case, the language provided can use the simple functional syntax found in today's OR-DBMSs – this means that there is no need to change SQL syntax on account of E-ADTs.

*Query Operators and Optimization:* An E-ADT can provide optimization routines that will translate a language expression into a query evaluation plan in its own evaluation algebra.

*Query Evaluation:* An E-ADT can provide routines to execute the optimized plan.

*Catalog Management:* An E-ADT can provide catalog routines so that schema information can be stored and statistics maintained on values of that E-ADT .

*Storage Management:* An E-ADT can provide multiple physical implementations of values of its type. Many existing OR-DBMSs already support this feature.

## 4.2 The $\mathcal{PREDATOR}$ DBMS

$\mathcal{PREDATOR}$[2] is a client-server OR-DBMS in which the server is a loosely-coupled system of E-ADTs. The core of the system is an extensible table in which E-ADTs are registered. The server is built on top of a layer of common database utilities that all E-ADTs can use. An important component of the utility layer is the SHORE Storage Manager [CDF$^+$94] library, which provides facilities for persistent storage, concurrency control, recovery and buffer management. The high-level picture of the system is shown in Figure 4.

Some of the basic types like integers do not support any enhancements. The figure shows two E-ADTs that do support enhancements: relations and sequences (i.e. time-series). An image E-ADT has also been added to

[2] $\mathcal{PREDATOR}$ stands for the PRedator Enhanced DAta Type Object Relational DBMS

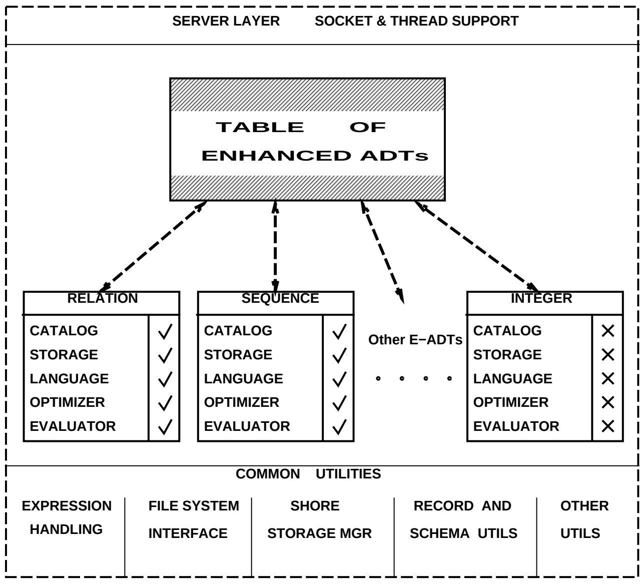


Figure 4: $\mathcal{PREDATOR}$ System Architecture

the system. A complex object like an image or a time-series can be a field within a relational tuple. An important feature of the system design is that E-ADTs are modular, with all E-ADTs presenting an identical internal interface. This serves a dual purpose:

- The development of optimizations for each E-ADT proceeds independently. The extensibility of the system would be compromised if each E-ADT had to be aware of all the other E-ADTs in the system.
- The interaction between the E-ADTs (especially between the SQL query processing engine and the individual E-ADT expressions) happens through the uniform internal interface, independent of the implementation of each E-ADT.

Due to space constraints, we do not discuss the detailed optimizations on the operations of each data type. More importantly, we only present a brief summary of the interaction mechanism between data types. A query is broken into components that correspond to sub-expressions of each E-ADT. Each sub-expression is treated as a declarative query and is parsed, optimized, and executed by its own E-ADT. It is difficult to build a DBMS with an extensible type system, while at the same time providing this rich functionality in each E-ADT. In fact, the E-ADT paradigm affects code design at many levels of the query processing engine. The details are in a longer version of this paper [SLR96a].

## 5 Related Work

This section is a brief survey of the related work in this area. [SLR96a] presents a more exhaustive description of related research and describes several exciting areas for future research enabled by the E-ADT paradigm.

**Postgres and Illustra:** The issues regarding support for ADTs in database systems were first explored in [SRG83] and [Sto86]. This led to the Postgres research DBMS [SRH90] and its commercial version, Illustra [Ill94]. The Postgres project explored issues dealing with the storage and indexed retrieval of ADTs. It also stressed that functions associated with ADTs could be expensive [Jhi88], and that special relational optimization techniques are necessary when such functions are present [Hel95]. The “ADT approach” described in this paper corresponds closely to Illustra’s technology. Currently, Illustra is being integrated into the Informix Universal Server [Inf96] and several modifications are being made to improve the parallel evaluation of ADT expressions [Ols96].

**Paradise:** The Paradise client-server DBMS [DKL+94] is developing ADT extensions for the parallel execution of operations on spatial, geographic, and scientific data. This work concentrates on issues of scalable parallelism and the use of tertiary storage for large ADTs. The importance of parallelism for ADTs arises from the large size of complex data types, and the high cost of operations on them. From the E-ADT viewpoint, parallelism is yet another possible benefit of declarative ADT expressions. Consequently, the Paradise work on parallel execution of ADT methods fits in neatly with the E-ADT paradigm. Paradise also implements some optimizations that allow ADT operations to retain intermediate results in memory.

There is a large amount of research in areas like object-oriented query processing [CD92, Cat94] that is related to our work. For example, Aberer and Fischer [AF95] observed that the semantic information about methods in an OO-DBMS can be used to optimize the query. Much of the recent work on extensible query optimization (like [MDZ93]) is also relevant. Several researchers have been exploring database support for collection types [Che96, Won94] and scientific data [MV93, WG93]. There has been an increasing interest in "special-purpose" database systems, which focus on data types or application domains of interest. This includes work on supporting queries over arrays, financial data, images, video, text, music, source code, etc. Our research impacts all this work, because our technology enables the domain specific query languages and optimizations developed in all these research projects to be integrated within a common database system. Finally, there are many similarities between the support for multiple data types, and the support of heterogeneous database systems. The loosely-coupled approach that we have taken encourages this comparison, and also opens up the possibility of building heterogeneous E-ADTs on top of the OLE-DB standard [Bla96].

## 6 Conclusion

The E-ADT paradigm is a novel approach to database systems design that "turbo-charges" the use of complex data types. E-ADTs are being built in the $\mathcal{PREDATOR}$ database system, and initial results indicate an order of magnitude increase in performance. Consequently, we believe that the next-generation of object-relational database systems should be based on E-ADTs.

## Acknowledgments

Praveen Seshadri was supported by IBM Research Grant 93-F153900-000 by an IBM Cooperative Fellowship, and by NASA Research Grant NAGW-3921. Miron Livny and Raghu Ramakrishnan were supported by NASA Research Grant NAGW-3921. Raghu Ramakrishnan was also supported by a Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563.

## References

[AF95] Karl Aberer and Gisela Fischer. Semantic query optimization for methods in object oriented database systems. In *Proceedings of the Eleventh IEEE Conference on Data Engineering, Taipei, Taiwan*, pages 70–79, 1995.

[Bla96] Jose Blakeley. Data Access for the Masses through OLE-DB. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data, Montreal, Canada*, pages 161–172, 1996.

[Cat94] R.G.G. Cattell. *The Object Database Standard:ODMB-93*. Morgan-Kaufman, 1994.

[CD92] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *sigmod92*, pages 383–392, 1992.

[CDF+94] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O. Tsatalos, S. White, and M.J. Zwilling. Shoring up persistent objects. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, pages 526–541, 1994.

[Che96] Mitch Cherniak. Form(ers) Over Function(s): The KOLA Reference Manual . Unpublished draft, Brown University, 1996.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. In *Proceedings of the Twenty Second International Conference on Very Large Databases (VLDB), Bombay, India*, pages 87–98, September 1996.

[DeW95] David J. DeWitt. DBMS: Roadkill on the Information Superhighway? Invited Talk: VLDB 95, 1995.

[DKL$^{+}$94] D.J. DeWitt, N. Kabra, J. Luo, J.M. Patel, and J. Yu. Client-Server Paradise. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB), Santiago, Chile*, September 1994.

[Hel95] Joseph M. Hellerstein. *Optimization and Execution Techniques for Queries With Expensive Methods*. PhD thesis, University of Wisconsin, August 1995.

[Ill94] Illustra Information Technologies, Inc, 1111 Broadway, Suite 2000, Oakland, CA 94607. *Illustra User's Guide*, June 1994.

[Inf96] Informix Software, Inc. *Informix and Illustra Merge to Form Universal Server*. Informix White Paper, http://www.informix.com/informix/corpinfo/zines/whitpprs/illustra/ifxillus.htm, February 1996.

[Jhi88] Anant Jhingran. A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, pages 88–99, 1988.

[Jon87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[LZ74] B. Liskov and S. Zilles. Programming with Abstract Data Types. In *SIGPLAN Notices*, April 1974.

[MDZ93] Gail Mitchell, Umeshwar Dayal, and Stanley Zdonik. Control of an Extensible Query Optimizer: A Planning-Based Approach. In *Proceedings of the Nineteenth International Conference on Very Large Databases (VLDB), Dublin, Ireland*, pages 517–528, 1993.

[MV93] D. Maier and B. Vance. A call to order. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Washington, DC*, 1993.

[Ols96] Mike Olson, 1996. Personal Communication.

[SLR96a] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Case for Enhanced Abstract Data Types. Submitted for Publication, 1996.

[SLR96b] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of the Twenty Second International Conference on Very Large Databases (VLDB), Bombay, India*, pages 99–110, September 1996.

[SRG83] M. Stonebraker, B. Rubenstein, and A. Guttman. Application of Abstract Data Types and Abstract Indices to CAD Data Bases. In *Proceedings of the Engineering Applications Stream of Database Week*, San Jose, CA, May 1983.

[SRH90] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[Sto86] Michael Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Proceedings of the Second IEEE Conference on Data Engineering*, pages 262–269, 1986.

[WG93] Richard Wolniewicz and Goetz Graefe. Algebraic Optimization of Computations over Scientific Databases. In *Proceedings of the Nineteenth International Conference on Very Large Databases (VLDB), Dublin, Ireland*, pages 13–24, 1993.

[Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, U.Pennsylvania, 1994.

# Storage and Retrieval of Feature Data for a Very Large Online Image Collection[‡]

Chad Carson and Virginia E. Ogle
Computer Science Division, University of California at Berkeley, Berkeley CA 94720
carson@cs.berkeley.edu, ginger@cs.berkeley.edu

**Abstract**

*As network connectivity has continued its explosive growth and as storage devices have become smaller, faster, and less expensive, the number of online digitized images has increased rapidly. Successful queries on large, heterogeneous image collections cannot rely on the use of text matching alone. In this paper we describe how we use image analysis in conjunction with an object relational database to provide both textual and content-based queries on a very large collection of digital images. We discuss the effects of feature computation, retrieval speed, and development issues on our feature storage strategy.*

## 1 Introduction

A recent search of the World Wide Web found 16 million pages containing the word "gif" and 3.2 million containing "jpeg" or "jpg." Many of these images have little or no associated text, and what text they do have is completely unstructured. Similarly, commercial image databases may contain hundreds of thousands of images with little useful text. To fully utilize such databases, we must be able to search for images containing interesting objects. Existing image retrieval systems rely on a manual review of each image or on the presumption of a homogeneous collection of similarly-structured images, or they simply search for images using low-level appearance cues [1, 2, 3, 4, 5]. In the case of a very large, heterogeneous image collection, we cannot afford to annotate each image manually, nor can we expect specialized sets of features within the collection, yet we want to retrieve images based on their high-level content—we would like to find photos that contain certain objects, not just those with a particular appearance.

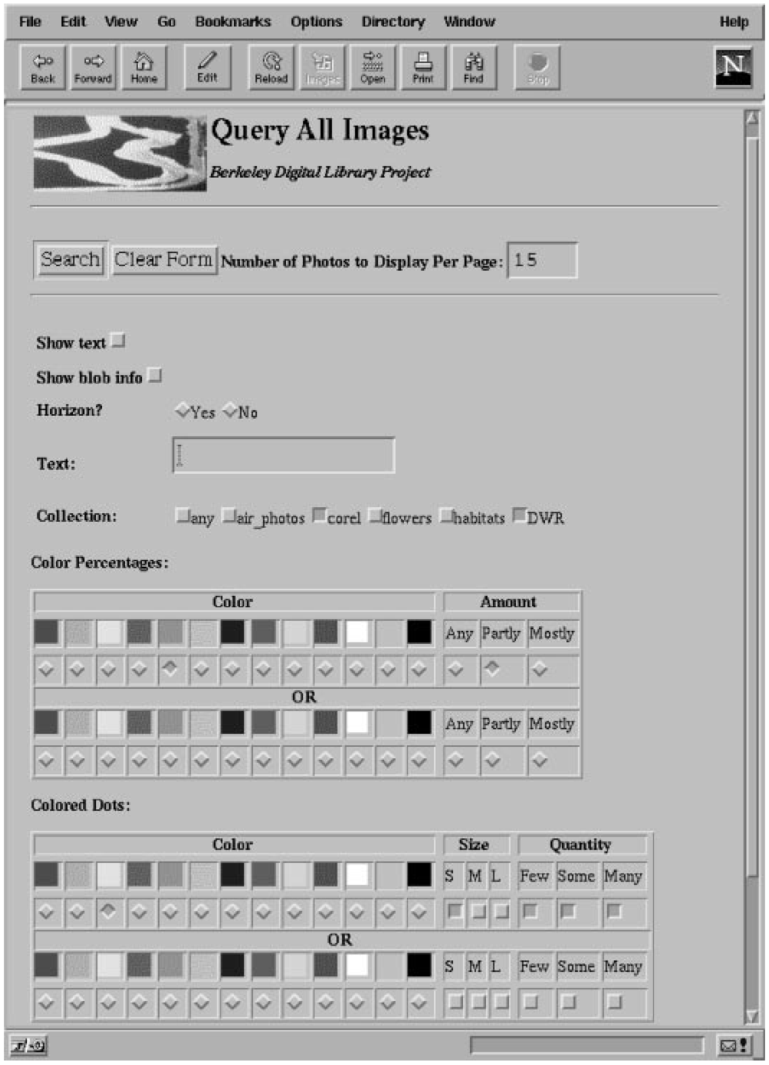
## 2 Background

The UC Berkeley Digital Library project is part of the NSF/ARPA/NASA Digital Library Initiative. Our goal is to develop technologies for intelligent access to massive, distributed collections comprising multiple-terabyte databases of photographs, satellite images, maps, and text documents.

In support of this research, we have developed a testbed of data [6] that as of this writing includes about 65,000 scanned document pages, over 50,000 digital images, and several hundred high-resolution satellite photographs.

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

[‡]This work was supported by an NSF Digital Library Grant (IRI 94-11334) and an NSF graduate fellowship for Chad Carson.



Query All Images

*Berkeley Digital Library Project*

Search | Clear Form | **Number of Photos to Display Per Page:** 15

**Show text** ☐

**Show blob info** ☐

**Horizon?** ◇Yes ◇No

**Text:**

**Collection:** ☐any ☐air_photos [✓ corel] ☐flowers ☐habitats [✓ DWR]

**Color Percentages:**

| Color | | | | | | | | | | | | | Amount | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | Any | Partly | Mostly |
| ◇ | ◇ | ◇ | ◇ | ◆ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◆ | ◇ |
| OR | | | | | | | | | | | | | | | |
| ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | Any | Partly | Mostly |
| ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ |

**Colored Dots:**

| Color | | | | | | | | | | | | | Size | | | Quantity | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | S | M | L | Few | Some | Many |
| ◇ | ◇ | ◆ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ✓ | ☐ | ☐ | ✓ | ✓ | ✓ |
| OR | | | | | | | | | | | | | | | | | | |
| ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | S | M | L | Few | Some | Many |
| ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

Figure 1: WWW query form set up for the "sailing and surfing" query.

This data is provided primarily by public agencies in California that desire online access to the data for their own employees or the general public. The testbed includes a large number of (text-based) documents as well as several collections of images such as photos of California native species and habitats, historical photographs, and images from the commercial Corel photo database. The image collection include subjects as diverse as wildflowers, polar bears, European castles, and decorated pumpkins. It currently requires 300 GB of storage and will require more than 3.4 TB when it is complete. Image feature data and textual metadata are stored in an Illustra database.

All data are now being made available online using access methods developed by the Berkeley Digital Library project. The data is accessible to the public at `http://elib.cs.berkeley.edu/` via forms, sorted lists, and search engines. Image queries can rely on textual metadata alone, such as the photographer's name or the photo's caption, or they can employ feature information about the image, such as color information or the presence of a horizon in the image (see figure 1).

## 3 Content-Based Querying

Most work on object recognition has been for fixed, geometric objects in controlled images (for example, machine parts on a white background), which is not very useful for image retrieval in a general setting such as ours. However, a few researchers have begun to work on more general object recognition [7].

The current focus of our vision research is to identify objects in pictures: animals, trees, flowers, buildings, and other kinds of "things" that users might request. This focus is the direct result of research by the user needs assessment component of the Digital Library project [8]. Interviews were conducted at the California Department of Water Resources (DWR), which is a primary source of the images used in the Digital Library project testbed as well as one of its main users. Employees were asked how they would use the image retrieval system and what kinds of queries they would typically make. The DWR film library staff provided a list of actual requests they had handled in the past, such as "canoeing," "children of different races playing in a park," "flowers," "seascapes," "scenic photo of mountains," "urban photos," "snow play," and "water wildlife."

As the user needs assessment team discovered, users generally want to find instances of high-level concepts rather than images with specific low-level properties. Many current image retrieval systems are based on appearance matching, in which, for example, the computer presents several images, and the user picks one and requests other images with similar color, color layout, and texture. This sort of query may be unsatisfying for several reasons:

- Such a query does not address the high-level content of the image at all, only its low-level appearance.
- Users often find it hard to understand why particular images were returned and have difficulty controlling the retrieval behavior in desired ways.
- There is usually no way to tell the system which features of the "target" image are important and which are irrelevant to the query.

Our approach is motivated by the observation that high-level objects are made up of regions of coherent color and texture arranged in meaningful ways. Thus we begin with low-level color and texture processing to find coherent regions, and then use the properties of these regions and their relationship with one another to group them at progressively higher levels [9]. For example, an algorithm to find a cheetah might first look for regions which have the color and texture of cheetah skin, then look for local symmetries to group some regions into limbs and a torso, and then further group these body segments into a complete cheetah based on global symmetries and the cheetah body plan.

## 4 Implementation

### 4.1 Finding Colored Dots

As a first step toward incorporating useful image features into the database, we have searched for isolated regions of color in the images. Such information can be useful in finding such objects as flowers and people.

We look for the following 13 colors in each image: red, orange, yellow, green, blue-green, light blue, blue, purple, pink, brown, white, gray, and black. We chose these colors because they match human perceptual categories and tend to distinguish interesting objects from their backgrounds [10].

We use the following algorithm to find these "colored dots":

1. Map the image's hue, saturation, and value (HSV) channels into the 13 perceptual color channels.
2. Filter the image at several scales with filters which respond strongly to colored pixels near the center of the filter but are inhibited by colored pixels away from the center. These filters find isolated dots (such as in a starry sky) and ignore regions that are uniform in color and brightness (such as a cloudy sky).

3. Threshold the outputs of these filters and count the number of distinct responses to a particular filter.

Responses at a coarse scale indicate large dots of a particular color; responses at finer scales indicate smaller dots. The number of dots of each color and size is returned, as is the overall percentage of each color in the image. A $13 \times 6$ matrix is generated for each image. Rows in the matrix represent the 13 colors that are identified. Six integers are associated with each color: the percentage of the image which is that color, and the number of very small, small, medium, large, and very large dots of that color found. (These sizes correspond to dots with radii of approximately 4, 8, 16, 32, and 64 pixels, respectively, in $128 \times 192$ pixel images.)

While these dot counts and percentages contain no information about high-level objects, they are a first step toward purely image-based retrieval. A number of combinations of the dot and percentage data yield interesting results; the following are a few examples:

| **Query** | **Percentages** | **Dots**[a] | **Text** | **Datasets** | **Precision**[b] |
|---|---|---|---|---|---|
| Sailing & Surfing (fig. 2) | blue-green $> 30\%$ | # VS yellow $\geq 1$ | — | Corel, DWR | 13/17 |
| Pastoral Scenes (fig. 3) | green $> 25\%$ AND light blue $> 25\%$ | | — | all | 85/93 |
| Purple Flowers (fig. 4) | | # S purple $> 3$ | — | all | 98/110 |
| Fields of Yellow Flowers | | # VS yellow $> 15$ | — | all | 63/74 |
| Yellow Cars | | # L yellow $\geq 1$ OR # VL yellow $\geq 1$ | "auto"[c] | all | 6/7 |
| People (fig. 5) | orange $> 1\%$ | # L pink $\geq 1$ OR # VL pink $\geq 1$ | — | Corel, DWR | 19/69 |

[a]The different dot sizes (very small, small, medium, large, and very large) are abbreviated VS, S, M, L, and VL, respectively.

[b]"Precision" is the fraction of returned images that contain the intended concept. "Recall," the fraction of images in the database containing the intended concept that are returned, is not a feasible measure in this case because we do not know how many instances of the intended concept are in the database.

[c]There are 132 "auto" images; restricting the query to images with large yellow dots reduces the number to seven.

## 4.2 Storage of Feature Data

Because of the size of the image collection and its associated metadata, we must use a database to manage both textual and image content information. Our chief priority is to store this data in such a way as to facilitate the fastest possible retrieval time in order to make rapid online browsing feasible. Therefore, we do not store the images themselves in the database, and we store metadata in a way that circumvents the need for joins on two or more tables. In addition, because image content analysis is time-consuming and computationally expensive, we do this analysis ahead of time and store the results in the database rather than using run-time functionality provided by the database. Another concern related to image analysis is the need to support continual development of new analysis techniques and new feature data. We want to be able to add new features and modify existing features painlessly as our vision research progresses. In this section we describe how our approach to storing image feature data meets these goals.

Each of the five image collections is stored in its own table with its own particular attributes. The collection of DWR images has 24 textual attributes per image, including a description of the image, the DWR-defined category, subject, and internal identification numbers. The wildflowers table contains 14 attributes per image such as common name, family, and scientific name. The Corel stock images have very little metadata: an ID number, a disk title such as "The Big Apple," a short description, and up to four keywords such as "boat, people, water,

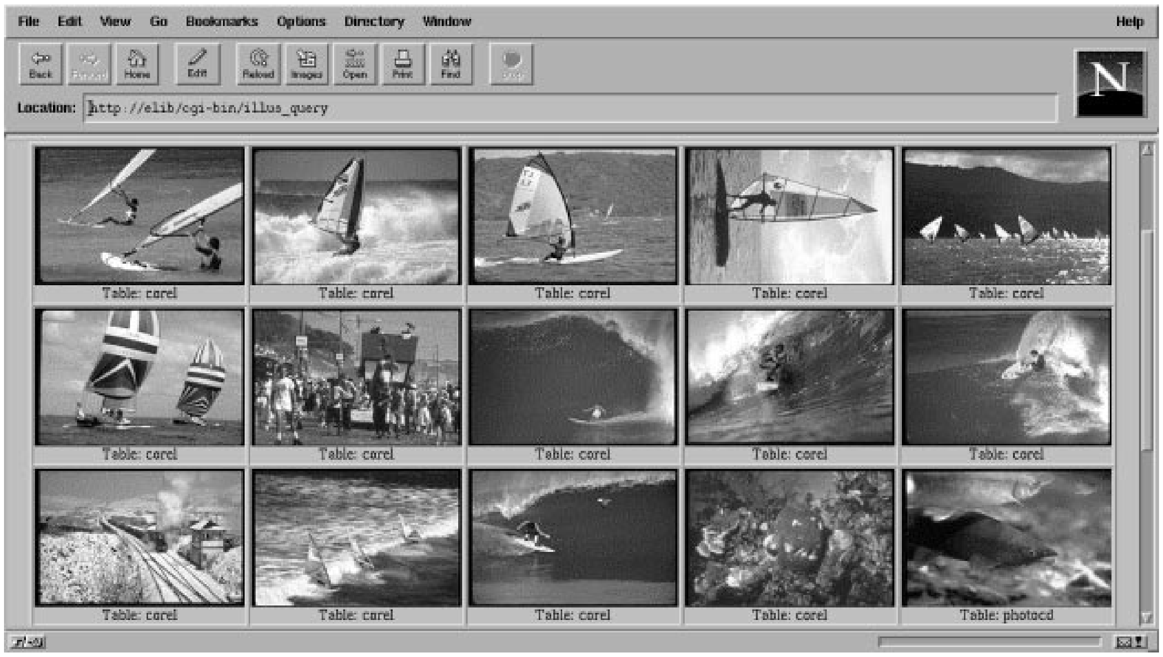

File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: http://elib/cgi-bin/illus_query
Table: corel
Table: photocd

Figure 2: Representative results for the "sailing and surfing" query. (Color images are available at `http://elib.cs.berkeley.edu/papers/db/`)

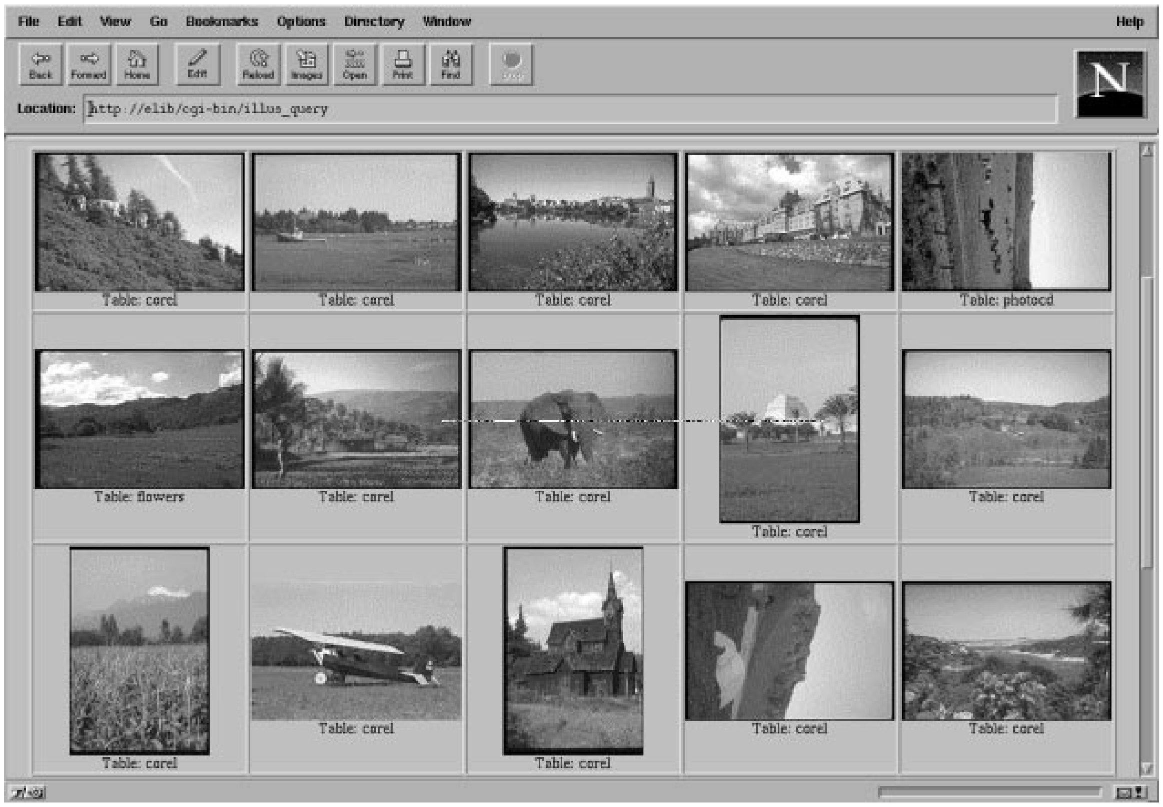

File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: http://elib/cgi-bin/illus_query
Table: corel
Table: photocd
Table: flowers

Figure 3: Representative results for the "pastoral" query.

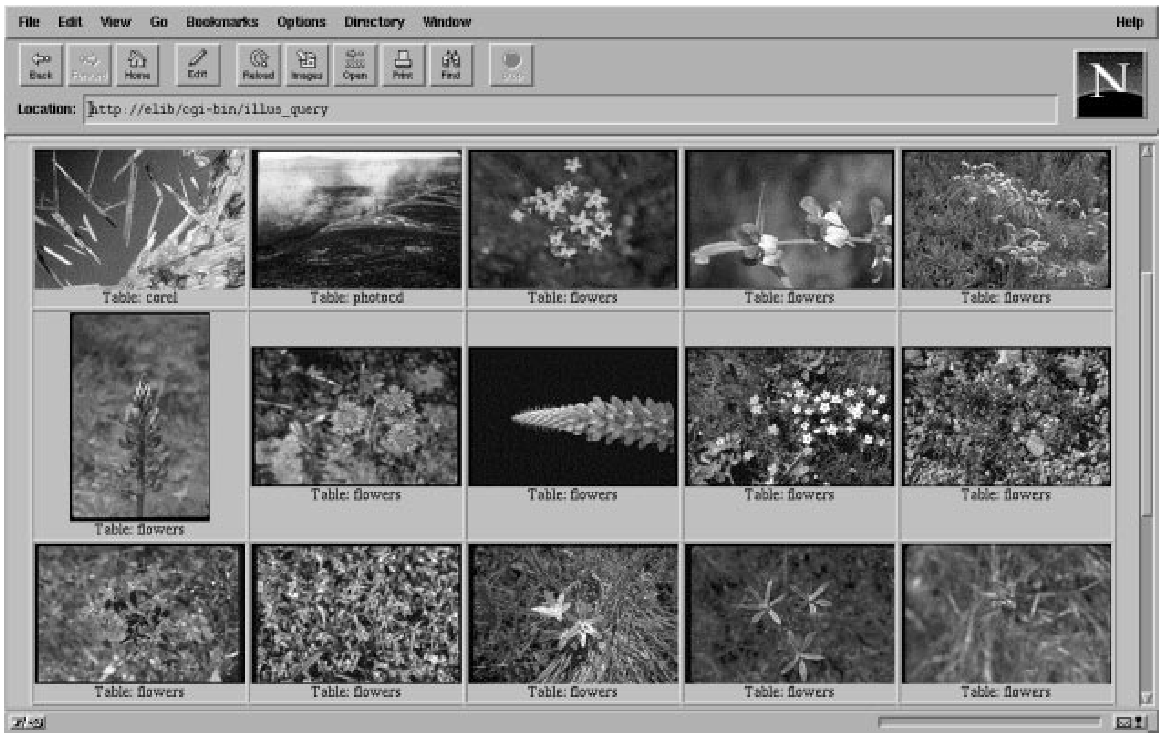

File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: http://elib/cgi-bin/illus_query
Table: corel
Table: photocd
Table: flowers

Figure 4: Representative results for the “purple flowers” query.

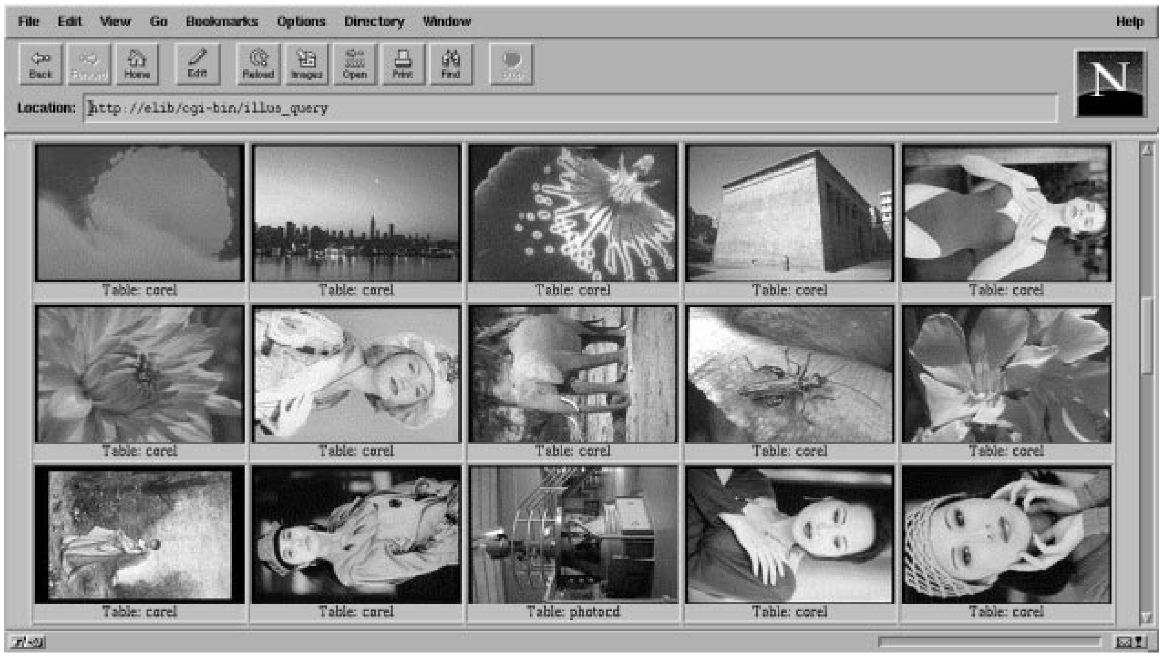

File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: http://elib/cgi-bin/illus_query
Table: corel
Table: photocd

Figure 5: Representative results for the “people” query.

mountain." The various image collections have very few textual attributes in common, other than a unique ID assigned by the Digital Library project and at least a few words of textual description from the data provider. Given the diversity of the overall collection and the likelihood of acquiring additional dissimilar image collections in the future, we do not want to support a superset of all image attributes for all the collections in one table. In addition, we have found that most users of our system want to direct a fairly specific query to a particular collection.

On the other hand, the addition of image feature data presents a more homogeneous view of the collection as a whole. Using image feature information to find a picture of sailboats on the ocean does not require any collection-specific information. Our approach is to support both text-based queries directed to a specific collection at a fine granularity ("find California wildflowers where common name = 'morning glory' ") and text/content-based queries to the entire collection ("find pictures that are mostly blue-green with one or more small yellow dots"). The separate tables for each collection are used for collection-specific queries, while collection-wide queries can be directed to an aggregate table of all images. This supertable contains selected metadata for every image in the repository: the collection name, the unique ID, a "text soup" field which is a concatenation of any available text for that image, and the feature data.

We have experimented with different ways of storing the types of feature data that have been developed so far, and we continue to try different techniques as new features are developed. Storage of Boolean object information, such as the presence or absence of a horizon in the image, is straightforward; we simply store a Boolean value for a "horizon" attribute. As our vision research proceeds and new kinds of objects can be identified, they can be concatenated onto an "objects" attribute string, so that each image has just one list—the objects that were found in that image. In this manner, we eliminate the need to record a "false" entry for each object not found in an image. This text string can be indexed, and retrieval is accomplished using simple text matching. However, more complex color and texture features, such as colored dot information, require careful planning in order to ensure fast retrieval, development ease, and storage efficiency. Interestingly, the complexity of the stored feature data is inversely related to the capability of the image analysis system: as computer vision systems become more adept at producing high-level output (e.g., "flower" instead of "yellow dot"), the question of storage and retrieval becomes simpler, because the level of detail of the stored information more closely matches the level of detail of desired queries.

**Storing Image Features as Text**

In general, we store image feature data as text strings, and we use text substring matching for retrievals. Dot information is stored in one text field per image. Any nonzero number of dots in an image is categorized as "few," "some," or "many" and stored in this field, separated by spaces. For example, a picture of a sky with clouds might have a few large white dots and a large amount of blue, so its dot field would be "`mostly_blue large_white_few`."

We have found that storing feature data as text yields the best results in terms of development ease, extensibility, and retrieval speed. We have experimented with other methods, such as storing dots as integer or Boolean values, and we have considered a compact encoding scheme for the feature data in order to save storage space and possibly cut down on retrieval time. But conservation of storage space is not a high priority for our project, and we have found that for fast retrieval time the use of text is satisfactory.

There are several advantages to using text instead of other data types. Most images have few significant objects and only two to five significant colors; each color typically has just a few of the dot attributes represented. The current implementation of dots would require 78 ($13 \times 6$) integer values, and most of them would be zero. Using one dots text string per image allows us to store only the features that are present in that image. This has an added benefit during the development stage, when vision researchers are testing their results on the image database—feature data can be concisely displayed in a readable form on the results page with little effort on the developer's part.

Using text also means that incremental changes to stored feature data do not require elaborate re-encoding or new attribute names. Text-based queries are simple to construct because there is just one dots field, as illustrated in the following example:

To find an image with "any kind of white dots" using text, we simply use wildcards in the select statement:

```
where dots like '%white%'
```

The equivalent integer expression requires five comparisons:

where VS_white $\geq$ 1 or S_white $\geq$ 1 or M_white $\geq$ 1 or L_white $\geq$ 1 or VL_white $\geq$ 1

Integer-based queries must be more carefully constructed to make sure that all possibilities are included in each expression. Such factors contribute to a faster development time if a text-based method is used, a bonus for a system like ours that is continually changing.

## 5 Future Directions

In the future we plan to investigate more efficient ways to store numerical feature data such as colored dots. However, as our image analysis research progresses, we expect to be able to use low-level feature information (shape, color, and texture) to automatically identify higher-level concepts in the images, such as trees, buildings, people, animals of all kinds, boats, and cars. As high-level information like this becomes available, the need to store low-level features like dots will decrease.

Currently most of the feature data we have developed is stored in a single table—the supertable that includes all the images in the collection. Although queries on this table can include text and can be directed to individual collections, no categorization of text is provided, because the primary purpose of the form is to make content-based queries. We plan to extend the content-based capability to the query forms for each individual collection so that users who know that particular collection can take advantage of the stored feature data. One collection that we think will benefit greatly from the use of content-based queries is the California wildflower collection. Users will be able to request pictures of a named flower in a particular color, such as "blue morning glories and not white morning glories," or even search for the names of flowers using color cues alone: "pink flowers with yellow centers" and "flowers with large purple blossoms."

## 6 Acknowledgments

We would like to thank David Forsyth, Jitendra Malik, and Robert Wilensky for useful discussions related to this work.

## References

[1] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, et al. Query by image and video content: the QBIC system. *IEEE Computer*, 28(9):23–32, Sep 1995.

[2] 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 frameworl for image management. In *Storage and Retrieval for Still Image and Video Databases IV*. SPIE, Feb 1996.

[3] U. Shaft and R. Ramakrishnan. Content-based queries in image databases. Technical Report 1309, University of Wisconsin Computer Science Department, Mar 1996.

[4] Michael Swain and Markus Stricker. The capacity and the sensitivity of color histogram indexing. Technical Report 94-05, University of Chicago, Mar 1994.

[5] A.P. Pentland, R.W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. *Int. Journal of Computer Vision*, to appear.

[6] Virginia E. Ogle and Robert Wilensky. Testbed development for the berkeley digital library project. *D-lib Magazine*, Jul 1996.

[7] J. Ponce, A. Zisserman, and M. Hebert. *Object representation in computer vision—II*. Springer LNCS no. 1144, 1996.

[8] Nancy Van House, Mark H. Butler, Virginia Ogle, and Lisa Schiff. User-centered iterative design for digital libraries: The cypress experience. *D-lib Magazine*, Feb 1996.

[9] J. Malik, D. Forsyth, M. Fleck, H. Greenspan, T. Leung, C. Carson, S. Belongie, and C. Bregler. Finding objects in image databases by grouping. In *International Conference on Image Processing (ICIP-96), special session on Images in Digital Libraries*, Sep 1996.

[10] G. Wyszecki and W.S. Stiles. *Color science: concepts and methods, quantitative data and formulae*. Wiley, second edition, 1982.

# Data Modeling and Querying in the PIQ Image DBMS [*]

Uri Shaft  Raghu Ramakrishnan

Computer Sciences Department, University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
email: (uri, raghu)@cs.wisc.edu

**Abstract**

*Image Database Management Systems (IDBMS) aim to store large collections of images, and to support efficient content-based retrieval. In this paper we explore the idea of* **image data modeling** *as a tool for describing image domains, with the twofold objective of guiding feature extraction and incorporating semantics into the extracted summaries of images. We discuss the implementation of these ideas in the* **PIQ** *image DBMS, and demonstrate that significant gains in the expressiveness of content-based queries can be achieved.*

Image Database Management Systems (IDBMS) aim to store large collections of images, and to support efficient content-based retrieval of the images. Like any other DBMS, an IDBMS has a *storage facility*, the data is arranged according to some *data model*, and it has a *query facility*.

The main IDBMS issue this paper explores is the data model and its impact on the *expressiveness* of the query language. The kind of queries we focus on are selection queries. In other words, we want to select a few images from the database, based on their content. Imagine a query facility that loads all the images from the database and checks each image to see if it fits the selection criterion. In all but extreme cases this query facility would be inefficient. The only other option is to have more information about the images stored in the database so it can be used for evaluating the selection criterion. We call this extra information a *summary*. When an image is stored in the database, its summary is stored as well. Queries are executed using only the information in the summaries. Images are retrieved only after we check that the summary conforms to the selection criterion. This means that any content information we want to query must be in the summary. Therefore, the expressiveness of any IDBMS query language is limited by the type of information in the summary.

Note that every existing IDBMS uses the idea of a summary for query optimization and execution. Some systems create the summaries automatically when an image is inserted into the database. Other systems need the help of a user in constructing the summary. User supplied input on a per-image basis has dire consequences on the system's ability to scale. One can't expect a user to provide input for many thousands of images. Therefore, a system capable of handling large collections of images must have automated summary construction (which includes *feature extraction*).

[*]This work was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award and NASA Research Grant NAGW-3921.

### 1.1 The Case for Data Modeling

Existing IDBMSs typically focus on a single domain of images. The type of information in the summary is determined at the time the system is built and can not be changed.

Images in a given specialized domain have much in common. This commonality is reflected in the type of information put in summaries. For example, the VIMS (or VIMSYS) [1, 3] system can handle only images of frontal views of single Human faces. The type of information in the summaries fits this image domain. A special feature extraction algorithm was constructed so that summaries can be created automatically. The query language is designed for this specific image domain as well. These kinds of systems have high development and maintenance costs. Every change to the system, such as including a new feature or a new type of query, needs expensive code development. Supporting a different image domain can only be achieved by building another system altogether.

Other systems consider the domain of all images. Any image can be inserted into the database, and all images are treated equally. Images inserted into the database may have nothing in common other than the fact that they are images. This means that the type of information that can be extracted automatically must be relevant to every possible image. This excludes any notion of semantic content of the information in the summaries. QBIC [4, 2] is an example for such a system. When an image is inserted with no user-supplied input, the system extracts two features of the image: a color histogram, and a condensed edge map that represents the layout of the image. The user may assist in identifying regions of the image (called *objects*). For each object, the system extracts its color histogram, its shape, and sometimes its texture. Any query language used in such a system has limited expressiveness because there is not much information in the summaries to work with.

PIQ differs from existing IDBMSs by being able to handle different specialized domains. We introduce the idea of *image data modeling* as a remedy to the problems listed above. Data modeling is used for describing the contents of images of a specific image domain. In [7, 6] we show how the PIQ data model is used for automated feature extraction. In this paper we demonstrate the expressiveness of a query facility built for the PIQ data model, and the scalability of PIQ.

## 2 The PIQ Data Model

In order to describe image collections to the IDBMS we need a *data description language* (DDL). This language has constructs that enable us to describe the common features of images in the same collection. Each valid description in the language is referred to as a *schema*. The collection of all valid schemas for the language is the *data model* of that language.

We describe the PIQ data model by demonstrating how schemas are defined using the PIQ DDL. For every image in a collection (with a specific schema) the system creates a summary. That summary's data type is defined by the schema. For example, suppose the schema for a collection of faces tells the system that the information about eye-color is stored in the summary. Then, for every image, the summary of the image will contain a value for eye-color (e.g. *blue*). The values of summaries are generated by an automated feature extraction facility (see [7, 6]).

It is important to note a fundamental aspect of PIQ: the DBMS does feature extraction and organizes the resulting summary information, in order to store, index and query this information efficiently. This results in a central tradeoff in the design of the data model: the richer the data model, the easier it is to describe a collection of images, but the harder it is to do appropriate feature extraction and summary organization.

### 2.1 What Is a PIQ Schema?

A schema is structured as a tree. Each node in the tree has a name and specific type. Furthermore, each node represents an *object* in the image. An object is a region of an image (i.e., a set of pixels).

The system maintains a set of *leaf node types*. Each leaf node type is associated with a feature extraction algorithm. For example, if we have an algorithm for extracting circular objects from an image, we can have a leaf node type for it (called *Circle*). A schema node with type *Circle* refers to a circular object in the image. The set of leaf node types is dynamic. Whenever we have some feature extraction algorithm we want to use, we can define a new leaf node type for it. Thus, the system's data model can be extended. For more details on DDL guided feature extraction see [7, 6].

Internal nodes of a schema tree represent operations done on the objects associated with the children of that node. Those types are:

*Union***.** The object referred to by a node whose type is *Union* is the union of the objects referred to by the node's children. Note that objects are sets of pixels, so a union of objects is a union of sets of pixels. For example, a schema for a face may have a *Union* node whose children represent *eyes, nose, mouth, hair* etc. Each child node represents a region in the image that corresponds to a part of the face. The *Union* node represents the region of the image that represents the entire face.

*Intersection* and *Difference***.** Defined the same as *Union,* differing only in the kind of set operation done on the objects the node's children refer to.

*Set***.** A *Set* schema node has exactly one child. The child node may have many valid objects to refer to in a single image. The set node indicates that all those objects need to be considered. The object associated with the *Set* node is the union of all the objects that can be associated with its child.

*Or***.** An *Or* schema has more than one child. Images in a collection may have valid objects in them only for a few of the node's children. Any object associated with a child of an *Or* node can be associated with the *Or* node itself. For example, the *Or* node of a schema may represent some *vehicle,* whereas its children may represent a *car,* a *bike,* a *bus* etc. If an image in the collection features a car, the object associated with the *Or* node is that car. Similarly, if an image in the collection features a bike or a bus, the object associated with the *Or* node is that bike or bus.

Each schema node contains attributes. Those attributes, like attributes in a relational data model, have names and types. PIQ maintains a set of types that can be used for defining attributes. This set may be extended by inserting new types into the system. However, there are some types that are always present in PIQ. These are: **Number**, **String**, **Boolean**, and **Region**. A **Region** type is a set of pixels. Each node has at least one attribute called *primary-region* (of type **Region**). This attribute's value is the object that node refers to. We also have an **Array** type for each of the above types (and for all other types inserted into the system). The array may have any number of dimensions and any size for each dimension.

The DDL statements defining schemas are:

- "**schema** *name* **has type** *type*." This declares a schema with a single node. The node has the name specified in *name* and the leaf node type specified in *type*.
- "**schema** *name* **is** *operator* **of** *list-of-names*." This declares a schema whose root node's name is *name*, and its type is *operator* (either *Union, Intersection, Difference* or *Or*). Its children are the schemas whose names are listed in *list-of-names*. The list of names actually lists pairs of names in the format: "prefix-1 : schema-name-1, ... , prefix-k : schema-name-k"
  We want the names of nodes in a schema to be unique. Since some of the names in the *list-of-names* can be duplicates, we make them unique by adding prefixes. The prefix is optional if there is no duplication of names. For example, suppose we have a schemas "A", "B". Suppose we want schema "C" to represent two objects of "A" and one of "B". The statement will be:
  "**schema** C **is** *Union* **of** first : A, second : A , B."
- "**schema** *name* **is** *Set* **of** *name1*". This declares a schema whose root node name's is *name,* and its type is *Set.* The child of that root node is the schema named *name1.*

Note that we identify a schema by the name of its root. The DDL statement for attribute definition is: "**attribute** *attr-name* **of node** *node-name* **of schema** *schema-name* **is** *expression*." This means that the node named *node-name* of the schema named *schema-name* has an attribute called *attr-name*. *expression* defines both the type of the attribute, and the method for computing it. More details about expressions are presented in Section 2.3.

## 2.2 What Is a PIQ Summary?

For each image we create a summary according to the schema of the image's collection. The summary is a tree structure that is very similar to the schema tree. Note that every sub-tree of a schema tree is also a schema. We build a summary tree for a specific image and schema using the following rules:

- Suppose the schema tree has only one node. The summary tree has only one node as well. For each attribute in the schema tree there is a value in the summary tree. That value is taken from the image. In particular, the value of the attribute *primary-region* has a **Region** type and refers to a set of pixels in the image.
- Suppose the schema tree has a root node of type *Union*. Each child of that root node is a root of a sub-tree (which is a schema). For each such sub-tree we construct a summary tree. The summary tree for the entire schema will have a root whose children are the roots of the summary trees constructed for the sub-tree schemas. Again, for each attribute in the schema root node, the summary root node will have a value. In particular, the value of *primary-region* will be the union of the values of *primary-region* of the children of the summary root node.
- Suppose the schema tree has a root node of type *Intersection* or *Difference*. We treat it the same as the *Union* case except that the rule for computing the value of *primary-region* changes to reflect the different set operation.
- Suppose the schema has a root node of type *Set*. The child of the root node is the root of a sub-tree (also a schema). We construct as many valid summary trees as possible for that sub-tree. The summary tree for the entire schema will have a root whose children are the roots of all the summary trees constructed earlier. Again, we have values for all attributes of the root node and the value of *primary-region* of the root node is the union of all values of *primary-region*'s in its children.
- Suppose a schema $S$ has a root node of type *Or*. Child $i$ of $S$ is the root of a schema $S_i$. For each $S_i$ we construct summary trees. Any summary tree that is valid for $S_i$ is also a valid summary tree for $S$.

## 2.3 Attributes and Expressions

Some attributes are defined by the user using the attribute definition statement described in Section 2.1. Other attributes are defined automatically. For each node in the schema tree the attribute *primary-region* is automatically defined. For each leaf node there may be other attributes that are automatically defined. This depends on the type of that leaf node. For example: a leaf node of type *Circle* may have attributes for the *position* of the center of the circle and for its *radius*.

When we construct a summary for an image with a specific schema, the automatically defined attributes get values from the feature extraction algorithm. For example, the position and radius of an object of a *Circle* node are determined by the algorithm that searches for the circle in the image. For leaf nodes, the extraction algorithm finds the value of *primary-region*. For internal nodes, we have rules that determine how to compute the value of *primary-region* from the values in its children (see Section 2.1 and 2.2).

The rules for computing the other attributes are given with their definition by an *expression*. An atomic expression is either a constant or a reference to a previously defined attribute. We construct more complex expressions from atomic expressions using function. When we construct a summary, the expression given for an attribute is a function whose variables are other attributes. When we evaluate the expression, we substitute the attribute references with their actual value to get the value for another attribute.

Expressions may be used to enforce some restrictions on a summary. The DDL statement is:
"**schema** *name* **is restricted by** *bool-expression*." This states that *bool-expression* is a boolean expression over the attributes of the schema *name*. This expression is called a *restriction*. A summary is valid only if it was constructed according to the rules in Section 2.2, and all the schema's restrictions evaluate to **true**.

An example for the use of a restriction is enforcing spatial relationships. Suppose we have a schema describing a face. A part of the schema can be stated as
"**schema** eyes **is** *Union* **of** left-eye : eye, right-eye : eye."
We assume that the schema for "eye" has position attributes (*x-position, y-position*). If we do not use restrictions and an image has two eyes in it then we have a problem. Each eye object in the image is a valid object for both "left-eye" and "right-eye". Therefore, we have four valid summaries for that schema. Obviously, we consider only one of the options as desirable. We can use the restriction
"**schema** eyes **is restricted by** x-position(left-eye) $<$ x-position(right-eye)."
The restriction claims that the left eye is positioned to the left of the right eye. Now, only the proper summary tree is valid.

### 2.4 An Example of a Schema and a Summary

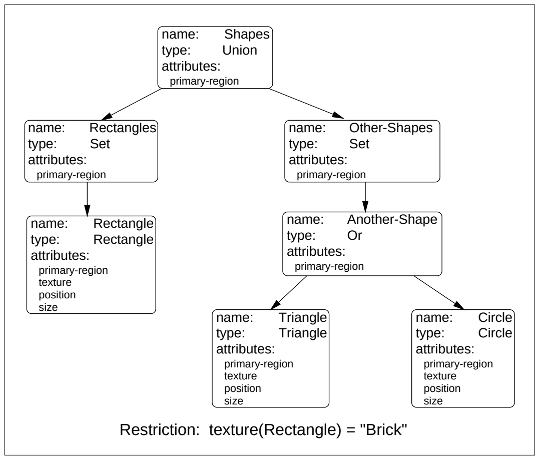


Figure 1: Example of schema.

Figure 1 is an example for a schema tree. The DDL definition of the schema is:
"**schema** Rectangle **has type** *Rectangle*."
"**schema** Rectangle **is restricted by** texture(rectangle) = 'BRICK'."
"**schema** Circle **has type** *Circle*."
"**schema** Triangle **has type** *Triangle*."
"**schema** Rectangles **is** *Set* **of** Rectangle."
"**schema** Another-Shape **is** *Or* **of** Circle, Triangle."
"**schema** Other-Shapes **is** *Set* **of** Another-Shape."
"**schema** Shapes **is** *Union* **of** Rectangles, Other-Shapes."
All the attributes are automatically specified in this case. This includes the *size*, *texture* and *position* of the leaf nodes.
Figure 2 shows an example of an image. Figure 3 shows a valid summary for that image with the schema in Figure 1.
The sub-images shown in each node of the summary tree in Figure 3 are illustrations representing the value of *primary-region* of that node. The values of other attributes are not shown. Note that there are only two rectangles in the summary instead of the four in the image. This happens because the schema has a restriction stating that the texture of a rectangle should be 'BRICK'. Only the two rectangles with a brick texture are presented in the summary.

## 3 Expressiveness of Queries

One can only ask queries about information that exists in the database, and is captured by the DBMS's data model. This point seems trivial, but it is the cause of major limitations to the leading IDBMSs such as QBIC [4, 2] and Photobook [5].

In the case of PIQ, one can create schemas, thereby determining what information in the images is important. In this case the true limit to the expressiveness of queries is the query language and query processing facility. Two main issues are still under research in the PIQ project. First, the issue of developing a good query language for the

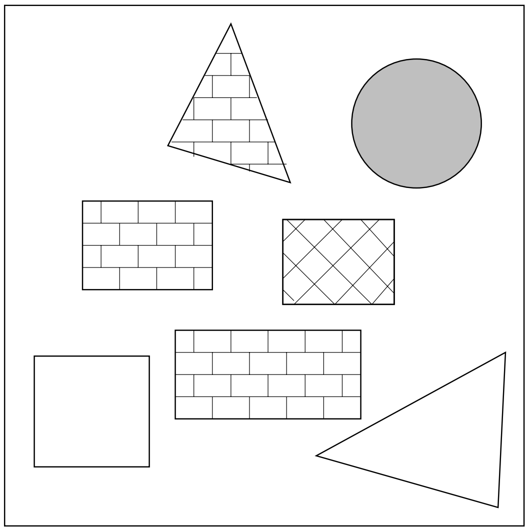


Figure 2: Example of image.

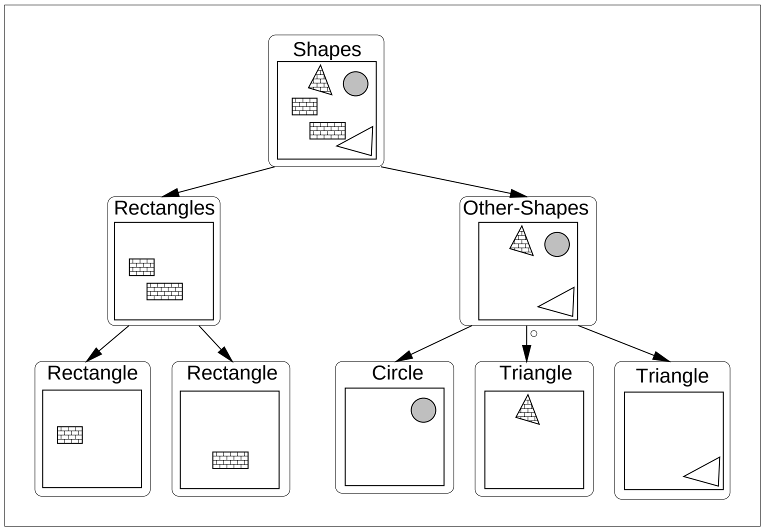


Figure 3: Example of summary.

PIQ data model. Second, how to optimize and process queries efficiently when the database consists of summary trees.

## 3.1 Representing Summary Trees in an Object-Relational Model

We can represent a schema as a set of relations, and a summary as a set of tuples in those relations, using the object-relational data model. Once we do that, queries can be posed in any object-relational query language (e.g. SQL variants). The main advantage is that the PIQ system can use a conventional DBMS for storing summaries and processing queries.

The representation of a schema as a set of relations is based on the idea that a tuple can represent a summary node, and the association between a parent node and its child can be captured by using foreign keys. For each node in the schema tree we create a relation with a unique name (which is the node's name). The attributes for each relation are the attributes of the schema node with the following additions:

- Image ID. An object ID for the image.
- Tuple ID. An object ID for the specific tuple.
- Parent ID (for every node except the root). For identifying which tuple represents the parent of this tuple in the summary tree.

Since we use object IDs and various data types that are not in the standard relational data model, we need an object-relational data model (for object IDs) with the ability to store large uninterpreted blobs (for arrays, regions and images).

Each node in a summary tree is converted into a tuple. The relation used for that tuple is the relation created for the schema node associated with the summary tree node. The summary tree node contains the values for the attributes, so those values are used for constructing the tuple. Furthermore, we get a unique object ID for the tuple and fill in the object IDs for the image and for the parent of this tuple.

We may not want all of the attributes to be used by the query facility. Some attributes may be computed only to be used for restrictions, but once the restrictions are evaluated, the values are not important for us. We can declare some attributes as *temporaries*. When creating the relations and tuples (or when storing the summary trees in any other way) we ignore those attributes. This can save a lot a storage space if the unwanted attributes are represented by large blobs such as arrays and regions.

## 3.2 Example of an Experiment with Airplane Pictures

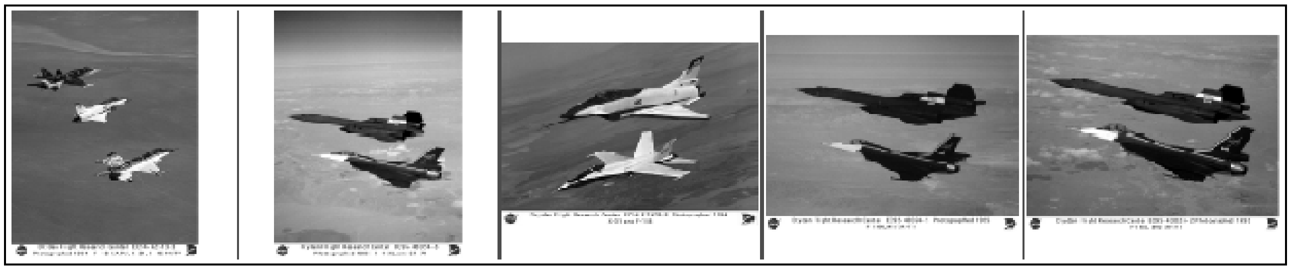


Figure 4: Result of query: “Images with at least two planes ordered by size of bottom plane.”

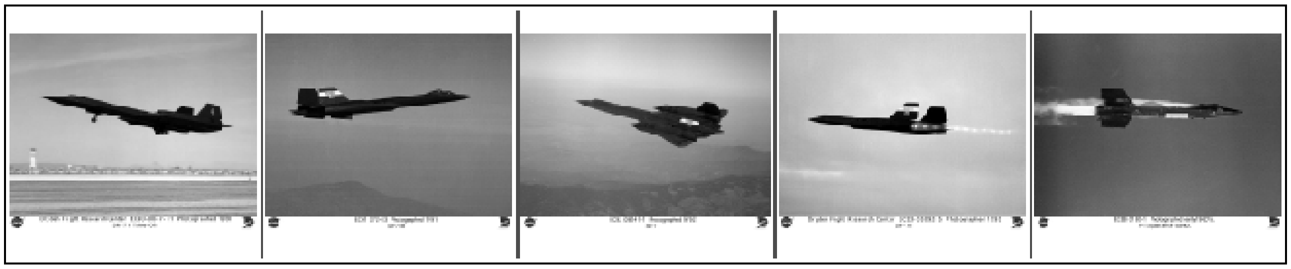


Figure 5: Similarity query. Best 5 results.

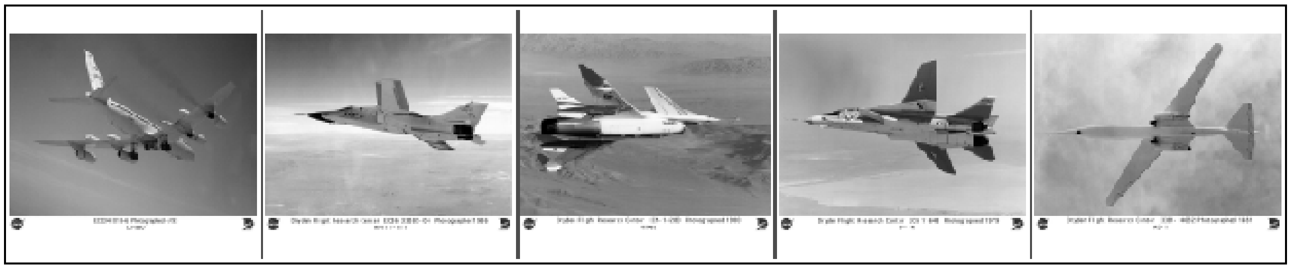


Figure 6: Same similarity query as in Figure 5. Worst 5 results.

The data set used in the experiment contains pictures of airplanes from the NASA Dryden Research Aircraft Photo Archive (available at `http://www.dfrf.nasa.gov/PhotoServer`). A very simple extraction algorithm that distinguishes between background and foreground was used for finding individual airplanes in the images. We created a schema leaf node type called *airplane* and associated it with that feature extraction algorithm. The attributes we get from the algorithm are *size, color, shape, orientation* and *position*. The schema is:
“**schema** airplane **has type** *airplane*.”
“**schema** airplanes **is** *Set* **of** airplane.”
We discarded the *primary-region* attributes. *shape* was represented by some algebraic moments of the *primary-region*. *color* is a color histogram of the airplane. *position* was broken down into the X and Y coordinates of the center of mass of *primary-region*. *orientation* is the angle that the main axis of *primary-region* has with the X axis. *size* is the number of pixels in *primary-region*.

All example queries in this section are expressible in SQL over summary relations. The queries are not easy to write in SQL. This is a major incentive to create a better query language.

Figure 4 shows the results of the query: “Find images with at least two planes ordered by size of bottom plane.” The order of results in the figure is from left to right. This query clearly demonstrates our ability to utilize spatial relationships (i.e., bottom) in conjunction with some other properties of objects (in this case *size*).

Figures 5 and 6 show the result of a similarity query. An example of an image is given (the image on the upper left in Figure 5). We want to find images that are similar to it according to a specified similarity metric. The metric in this case is: images with the same number of airplanes in them as the example image. The ordering of the images is: first order by how close the color of the airplane is, then order by the size and shape and orientation. The results in Figure 5 are ordered from left to right.

Note that most systems have built-in notions of similarity. It may be useful to reduce the amount of work needed for specifying a query by using a default similarity metric. However, it limits the user's ability to specify his/her requirements for the query.

## 4 Conclusions and Future Work

We described the idea of image data modeling and its application in the PIQ IDBMS. We showed that a very powerful query facility is feasible when using a non-trivial data model.

The PIQ system is extensible and scalable. The system's data model can be extended by defining new leaf node types for schemas. This enables the user to have automated feature extraction for many image domains by including specialized software developed by the Computer Vision community. Automated feature extraction makes the system scalable: the number of images inserted into the database can be very large. The reason is that we do not need to have user's input on a per-image basis, and summaries can be indexed and queried using scalable DBMS techniques.

Our future work focuses on developing a good query facility. This includes a query language that is both powerful and easy to use. It also includes solving the problem of query optimization and processing for summary trees.

## References

[1] Jefferey R. Bach, Santanu Paul and Ramesh Jain. A visual information management system for the interactive retrieval of faces. *IEEE Transactions on Knowledge and Data Engineering,* 5(4):619–628, August 1993.

[2] C. Faloutsos et al. Efficient and Effective Querying by Image Content. *Journal of intelligent information systems,* 3(3):231–262, 1994.

[3] Amarnath Gupta, Terry E. Weymouth and Ramesh Jain. Semantic Queries with Pictures : The VIMSYS Model. *17th International Conference on Very Large Data Bases,* Barcelona, Spain, September 1991.

[4] W. Niblack et al. The QBIC project: Querying images by content using color, texture, and shape. In Wayne Niblack, editor, *SPIE vol. 1908: Storage and Retrieval for Image and Video Databases,* pages 173–187, San Jose, CA, February 1993.

[5] A. Pentland, R. W. Pickard and S. Scaroff. Photobook: Tools for content-based manipulation of image databases. In Wayne Niblack and Ramesh C. Jain, editors, *SPIE vol. 2185: Storage and Retrieval for Image and Video Databases II,* pages 34–47, San Jose, CA, February 1994.

[6] Uri Shaft and Raghu Ramakrishnan. Content-Based Queries in Image Databases. Technical report 1309, Computer Sciences Department, University of Wisconsin-Madison, March 1996.

[7] Uri Shaft and Raghu Ramakrishnan. Data modeling and feature extraction management in image databases. In C.-C. J. Kuo, editor, *SPIE volume 2916: Multimedia Storage and Archiving Systems,* Boston, MA, November 1996.

# An Optimizer for Heterogeneous Systems with NonStandard Data and Search Capabilities

Laura M. Haas[*] Donald Kossmann[†] Edward L. Wimmers[‡] Jun Yang[§]
IBM Almaden Research Center
San Jose, CA 95120

**Abstract**

*Much of the world's nonstandard data resides in specialized data sources. This data must often be queried together with data from other sources to give users the information they desire. Queries that can span several specialized sources with diverse search capabilities pose new challenges for query optimization. This paper describes the design and illustrates the use of a general purpose optimizer that can be taught about the capabilities of a new data source. Our optimizer produces plans of high quality even in the presence of nonstandard data, strange methods, and unusual query processing capabilities, and makes it easy to add new sources of standard or nonstandard data.*

## 1 Introduction

Much of the world's nonstandard data resides in specialized data sources. For example, videos are typically found in video servers that know how to manage quality of service issues; chemical structures are found in specialized chemical structure "databases"; and text is found in a variety of information retrieval and file systems. Many of these sources have specialized query processing capabilities. Some videos are now indexed by scene changes; chemical structure databases support substructure and similarity search; and information retrieval systems support content search of various degrees of sophistication. Furthermore, this data, though in specialized sources, does not exist in isolation. It often must be combined with other data from other sources to give users the information they desire.

Database middleware systems offer users the ability to combine data from multiple sources in a single query. Several projects are currently working on middleware to bridge sources of nonstandard data types [SAD$^+$94, LP95, PGMW95]. Queries that can span several specialized sources with diverse search capabilities pose new challenges for query optimization in these projects. With multiple sources, the possible ways to execute any given query are many. Special-purpose optimizers that each understand a particular set of data sources could be built,

[*] *laura@almaden.ibm.com*
[†] Current address: University of Passau, 94030 Passau, Germany; *kossmann@db.fmi.uni-passau.de*
[‡] *wimmers@almaden.ibm.com*
[§] Current address: Stanford University, Stanford, CA 94305; *junyang@db.stanford.edu*

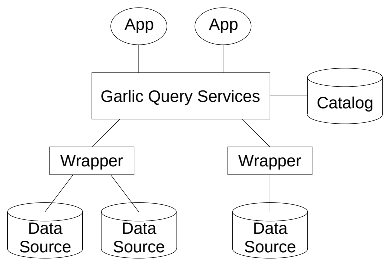


Figure 1: Garlic System Architecture

but due to the range of nonstandard data types and systems storing them, and the variety of combinations that users need, this is not a practical approach.

In the Garlic project [C$^+$95], we are building a general purpose optimizer that can be taught about the capabilities of a new data source. We make it easy to add new sources of non-standard data and to exploit the special query processing abilities of sources. Our approach produces plans of high quality, while making it possible to include easily a broad range of data sources. In addition, we can both extend the system with new sources and evolve the descriptions of sources to capture further capabilities at any time.

This paper illustrates how our optimizer helps users relate together data of nonstandard types, and exploit the special query capabilities that may be associated with them. In the next section, we briefly discuss the middleware model we assume, and sketch our approach to optimization. A detailed description of our approach can be found in [KHWY96]. In Section 3, we give an example of how our optimizer would work for a query of chemical structure data. Section 4 summarizes the paper and includes a quick discussion of related work.

## 2 Optimization in a Middleware Architecture

The architecture of Figure 1 is common to many heterogeneous database systems, including TSIMMIS [PGMW95], DISCO [TRV96], Pegasus [SAD$^+$94], DIOM [LP95], HERMES [ACPS96] and Garlic [C$^+$95]. Data sources store data and provide functions to access and manipulate their data. These are pre-existing systems not to be disturbed by the middleware. A *wrapper* protects the data source from the middleware, and enables the middleware to use its internal protocols to access the source. The wrapper describes the data and capabilities of a source in a common data model. In Garlic, data is described using an object-oriented model based on the ODMG standard [Cat96, C$^+$95]. Collections of objects are defined, and can serve as the targets of queries. Objects were chosen for their ability to model the full range of nonstandard data encountered. Wrappers provide methods to access the attributes of objects, and specialized search capabilities of the source are also encapsulated as methods. In this paper, we are concerned with middleware systems that promote “thin” wrappers, that is, systems in which the middleware does not mask the differences among sources, and instead, exploits the specialized capabilities of those sources.

The query services component of these systems must handle nonstandard data and query capabilities in planning and executing queries. Query services typically consist of a query language processor and an execution engine. The query processor obtains an execution plan for the query through some sequence of parsing, semantic checking, query rewrite, and query optimization, drawing on a system catalog to learn where data is stored, what wrapper it is associated with, its schema, any available statistics, and so on. The execution engine passes the sub-queries identified in the plan to the wrappers and assembles the final query result. A key feature of Garlic is that each wrapper provides a description of its source’s query capabilities. This description is used by Garlic’s query optimizer to create a set of feasible plans and to select “the best” for execution.

Garlic follows a traditional, dynamic programming approach to optimization [SAC$^+$79]. Plans are trees of Plan Operators, or *POPs*, characterized by a fixed set of plan *properties*. These properties include *Cost*, *Ta-*

*bles*, *Columns*, and *Predicates*, where the latter three keep track of the collections and attributes accessed and the predicates applied by the plan, respectively. The enumerator builds plans for the query bottom-up in three phases, applying pruning to eliminate inefficient plans at every step. In the first phase, it creates plans to access individual collections used in the query. In the second phase, it iteratively combines plans to create join plans, considering all *bushy* join orders. Bushy plans are particularly efficient for distributed systems in many situations. Finally, the enumerator adds any POPs necessary to get complete query plans. The winning plan is chosen on the basis of a cost model that takes into account local processing costs, communication costs, and the costs to initiate a sub-query to a data source. The cost model should include the costs of expensive methods and predicates [HS93, CG96].

Garlic uses grammar-like *STrategy Alternative Rules* (*STARs*) as in [Loh88] as the input to enumeration. Each wrapper defines its own set of POPs, to describe the query processing capabilities it exports. The fixed set of properties used to describe POPs allows Garlic to handle plans that are (in part) composed of POPs whose specifications are unknown to Garlic because they are defined by wrappers. Every wrapper also has a set of STARs, which construct plans that can be handled by that wrapper, using the wrapper's POPs. Likewise, Garlic has STARs which construct plans using Garlic's POPs. The Garlic STARs use wrapper plans as building blocks, combining them to generate full plans for the query. The Garlic STARs require certain properties of the wrapper plans in order to combine them; if there are no plans with the necessary properties, other Garlic STARs are invoked to add Garlic POPs to existing plans until the correct properties are achieved. When a STAR is applied during enumeration, all properties of the resulting plans are computed.

We call the topmost non-terminal symbols of the grammar *roots*. While STARs and POPs are defined for every wrapper individually, Garlic defines a fixed set of roots with fixed interfaces, corresponding to the various language functions it supports. There are roots for *select*, *insert*, *delete*, and *update*. For example, an `AccessRoot` STAR models alternative ways to access a collection of objects from a data source. Wrappers may provide STARS for some or all of the roots. A wrapper must export at least one `AccessRoot` STAR if data in its data sources are to be accessible in queries. Since no data are currently stored in Garlic itself, there are no `AccessRoot` STARs defined for Garlic. Garlic, however, has several `JoinRoot` STARs that model the alternative ways to execute a join in Garlic's query engine. In addition, Garlic defines a `FinishRoot` STAR to complete plans, by adding POPs to enforce properties not yet taken care of by any wrapper or Garlic STARs. Further details of our approach to optimization can be found in [KHWY96].

Once the optimizer chooses a winning plan for the query, the plan is translated into an executable form. Garlic POPs are translated into operators that can be directly executed by the Garlic execution engine. Typically each Garlic POP is translated into a single executable operator. By contrast, an entire subtree of wrapper POPs is usually translated into a single query or API call to the wrapper's underlying data source. Wrappers are, however, free to translate POPs in whatever way is appropriate for their system.

## 3 Example: Optimization of Pharmaceutical Queries

In this section, we show how queries in Garlic are optimized, using as an example a pharmaceutical research application. In collaboration with chemists at the Almaden Research Center and a large pharmaceutical company, we are integrating a number of data sources including RDBMSs storing the results of biological assays, chemical structure databases, and text search engines containing paper abstracts and patents. Although many queries in this environment are quite complex and involve several data sources, we focus for brevity on queries to a single, nonstandard data source, a chemical structure database. We present POPs and STARs for this data source, and, for a sample query, show alternative plans that could be generated using those STARs in combination with Garlic's STARs. Many STARs, even for unusual data sources, have the same simple pattern, making them easy to write. Furthermore, we show that the differences in cost of alternative plans can be large, even when all data is in a single data source; hence it is important to have an optimizer that can enumerate all alternative plans. This is as true in an environment with a diversity of sources storing nonstandard data as in a standard relational environment.

## 3.1 STARs for a Chemical Structure Database

The chemical structure database maintains collections of molecules. This data source (and its wrapper) can handle two kinds of queries: *substructure* queries, which return the key (i.e., the *l_number* field) of all molecules that contain a certain substructure, say, "$CC(C)C$"[1], and *similarity* queries. Given a sample molecule, similarity queries compute a score in the range of [0,1] for every molecule, measuring how similar each is to the sample molecule; the query returns all molecules of a collection ordered by this score. Thus, the first kind of query corresponds to the application of a boolean predicate (i.e., filter) to a collection of molecules whereas the second kind of query corresponds to ranking molecules of a collection in the same way as is done for web pages in a WWW search engine or for images in an image processing system.

In addition to these two kinds of queries, the wrapper of the chemical structure database can handle method calls. Method calls can be made to fetch the value of an attribute of a molecule, to determine whether a specific molecule (specified by its *l_number*) contains a certain substructure, or to determine the similarity score of a specific molecule as compared to a sample molecule.

To model the ability to execute substructure queries, our chemical wrapper defines the STAR shown in Figure 2. The STAR is an `AccessRoot` STAR used in the first phase of plan enumeration for a Garlic *select* query (i.e., a read-only query). Like all `AccessRoot` STARs, it takes three parameters: (1) $T$, which specifies a collection of molecules used in the query; (2) $C$, the set of all expressions used in the various clauses of the query; and (3) $P$, the set of predicates found in the *where* clause of the query.

$$\texttt{AccessRoot}(T, C, P) = \forall p \in P : \textbf{M\_Select}(T, p)$$

*Conditions:* $p$ is a *substructure* predicate

Figure 2: substructure `AccessRoot` STAR

The STAR generates a list of alternative plans to retrieve the molecules of collection $T$. Each plan consists of a single *M_Select* POP which models filtering the collection by a substructure predicate. The semantics of an *M_Select* POP are not known to the Garlic optimizer, but the wrapper sets the plan properties to inform the optimizer which parts of the query are handled by this plan. In this case, the *Column* property would be set to $\{$*l_number*$\}$ specifying that only the key of the molecules is returned; the *Predicate* property would be set to $\{p\}$ specifying that the substructure predicate $p$ has been applied, and of course, the wrapper's cost model would be consulted in order to compute the estimated *cost* property of the *M_Select* plan; Garlic's optimizer uses the *cost* property in order to carry out pruning and to determine the winning plan. Ultimately, the *M_Select* POP of the winning plan would be translated by the wrapper at execution time into a sub-query to the chemical structure database.

The *M_Select* POP can only apply a single substructure predicate at a time. This is because the chemical structure database can only apply a single predicate at a time (this is a good example of a data source with both nonstandard data and query capabilities). For queries that have several substructure predicates, the STAR will enumerate a separate, alternative access plan for every individual predicate. Depending on the selectivity of the various predicates, and the expense of a method call to test substructure, these alternative access plans can have quite different costs. For queries that have no substructure predicates, the STAR will fail and return no plan (other STARs would be applicable for such queries). Furthermore, the *M_Select* POP does not compute scores for similarity search; this is because the chemical structure data source cannot handle substructure predicates

[1] All chemical formulas in this paper are presented in *SMILES* notation, which is emerging as a standard description language for chemical structures.

and similarity search in a single query. Finally, the conditions on the STAR guarantee that the wrapper will only be given substructure predicates; this is because this data source cannot evaluate predicates such as *l_number* $<$ "M38". These predicates would have to be evaluated by Garlic after retrieving the raw data from the source.

$$\texttt{AccessRoot}(T, C, P) = \forall e \in C : \textbf{M\_Rank}(T, e)$$

*Conditions:* $e$ is a *similarity* expression

Figure 3: Similarity `AccessRoot` STAR

The STAR of Figure 3 models that similarity searches can be carried out by the chemical structure database. Again, the STAR is an `AccessRoot` STAR, and it takes the same three parameters as the substructure `AccessRoot` STAR. Rather than generating alternative plans with an *M_Select* POP, this STAR generates alternative plans with an *M_Rank* POP. To model the execution of a similarity search, the properties of the *M_Rank* POP are set in the following way. The *Column* property is set to contain *l_number* and $e$, the scoring expression on which the similarity search is based. The *Predicate* property is set to $\emptyset$ as no predicates can be evaluated by the wrapper or data source during a similarity search. Unlike substructure queries, similarity searches return results ordered by score; accordingly the *M_Rank* POP sets the *Order* property to $e$ DESC. Finally, of course, a special cost function would be invoked to compute the *cost* property of *M_Rank* plans. As for substructure predicates above, only one score expression can be evaluated as part of a similarity search at the data source.

Both STARs of the chemical wrapper are very simple; nevertheless, they are sufficient to construct good plans for queries over molecules, as will be shown in the next subsection. In general, most wrapper STARs are simple because wrapper STARs model "what" sub-queries can be handled by a wrapper and its data sources; wrapper STARs need not model "how" these sub-queries are actually executed. For example, the STARs of Figures 2 and 3 do not enumerate alternative plans using different indexes since access path selection is carried out autonomously by the chemical structure database system. Since Garlic STARs, like the rules for plan generation of any standard dbms, must specify how Garlic executes its portion of a plan, they are generally significantly more complex. However, Garlic's optimizer is completely general; in other words, Garlic STARs are written once (by the developers!), and no new Garlic STARs are needed to add a new data source, only the simpler wrapper STARs.

## 3.2 Example Query and Plans

In the following, we will show how wrapper STARs work together with Garlic's pre-defined STARs to generate alternative plans for a query. As discussed in the previous section, the chemical wrapper and its data source can do only one thing at a time: either apply a substructure predicate or run a similarity search. Garlic makes it possible to ask queries that cannot be evaluated natively by the data source. For example, Garlic can answer queries with both a substructure predicate and a similarity search, by issuing method calls or executing joins to combine partial query results returned from the chemical wrapper. To illustrate this point, we use a query that asks for poisonous molecules containing "$c1ccccc1$" (benzene), that are similar to a "$CC(C)C$" molecule (isobutane).

```
select m.l_number, m.similarTo("CC(C)C") as relevance
from PoisonousMolecules m
where m.containsSubStructure("c1ccccc1")
order by relevance desc
```

Figure 4 shows three alternative plans for this query. In the first phase of enumeration, the Garlic optimizer uses the STARs of the previous section to generate *M_Select* and *M_Rank* POPs. In the second and third phases, the Garlic STARs are fired to complete the plans. The POPs generated by the Garlic STARs are denoted by "G_" in Figure 4 and are fairly self-explanatory. The *G_Pushdown* POP submits a sub-query to a wrapper and receives results from the wrapper. *G_Fetch* fetches so-far unretrieved or uncomputed columns (e.g., scoring expressions) using method calls. *G_Filter* filters out rows, applying methods as needed to do so. *G_Join* is a logical join operator that combines two input streams and *G_Sort* orders the incoming stream of values.

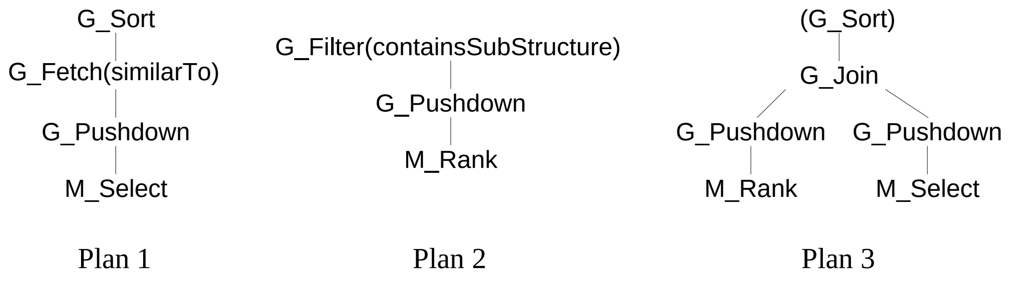


Figure 4: Alternative Query Evaluation Plans

Plan 1 of Figure 4 would be executed as follows: run the substructure query in the chemical structure source, then for each qualifying molecule call a method to compute its similarity score, and sort the resulting ⟨molecule, score⟩ pairs by score. Plan 2 carries out the similarity search in the chemical structure source, and then filters out molecules that contain a "$c1ccccc1$" structure, using *containsSubStructure* method calls to compute the truth value of the predicate. This second plan does not require a final sort in Garlic as the molecules are already in the right order as a result of the similarity search. Plan 3 executes both a similarity search and a substructure query in the molecule database and then does a self-join in Garlic to assemble the final query result; depending on the join method used to implement the G_Join POP final sorting of the molecules might be necessary. Plan 3 is generated by special Garlic self-join STARs that combine different access plans produced by wrapper STARs.

As each POP of each plan is added, its cost property would be computed. The cost model of the chemical wrapper would be consulted to estimate the cost of the *M_Select* and *M_Rank* sub-queries (i.e., to compute the *cost* property of these POPs), and Garlic's cost model would be consulted to estimate the cost of joins and sorting in Garlic, the overhead of method calls, and the communication costs. Clearly, the differences in cost between the three plans can be large so that it is important to enumerate and cost out all three of them. If, say, 10,000 poisonous molecules are stored in the chemical structure data source, Plan 2 would induce the overhead of 10,000 substructure method calls while executing the *G_Filter* in order to find the qualifying molecules composed of benzene. The overhead of method calls can be substantial in this environment because every method call requires sending a request to the data source and possibly setting up internal structures in the data source to evaluate the request. Depending on the number of benzene molecules (i.e., the selectivity of the substructure predicate) significantly fewer method calls are issued for Plan 1: if, say, 1,000 benzene molecules exist, then only 1,000 *similarTo* method calls are required as part of the *G_Fetch* operation of this plan. The reduction in the number of method calls comes, however, at the cost of sorting the 1,000 resulting molecules in Garlic. The use of method calls is completely avoided in Plan 3, but Plan 3 requires paying the price of an additional join in Garlic.

## 4 Conclusions

The optimizer described and illustrated in the previous sections has been implemented as part of the Garlic project. Further details of its implementation can be found in [KHWY96]. This optimizer is based on traditional, well-understood optimization technology, and can handle queries to data sources with standard and nonstandard data and search capabilities. To accomplish this, our optimizer extends the traditional approach by allowing strategy alternative rules and cost models to be defined separately for each wrapper. Because rules for wrappers are typically quite simple, and because rules for different wrappers are defined separately, the system is easily extensible, and can support a broad range of wrappers for data sources with diverse and specialized capabilities. Since the optimizer is cost-based and enumerates the entire space of feasible plans, it finds good plans even in the presence of nonstandard data, strange methods, and unusual query processing capabilities.

Only recently have other projects addressed the problem of optimization in this environment [FRV95, Qia96, PGH96, TRV97]. Most take the approach of decomposing, or rewriting, the query into wrapper-specific pieces, and heuristically pushing down maximal pieces to the wrapper, though of course the actual means of decomposition varies. An exception is the optimizer of DISCO, which also does a cost-based optimization [TRV97]. However, DISCO enumerates plans as though all wrappers could handle any query, then uses wrapper grammars to filter out impossible plans. The Garlic optimizer, by contrast, enumerates only valid plans.

We have chosen a Selinger-style, dynamic programming approach to optimization because of its proven efficiency in finding good plans. We have followed Lohman's STAR framework, because of the advantages it offers in terms of extensibility. We believe that our work extends these benefits into the heterogeneous environment of sources of nonstandard data in a simple and compelling way. New wrappers of nonstandard sources are typically up and running in Garlic in a matter of days, and initial performance results reported in [KHWY96] indicate that the dynamic programming approach to optimization will work as well for this environment as it has worked in the past for standard relational data.

## References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 137–148, Montreal, Canada, June 1996.

[C$^+$95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, March 1995.

[Cat96] R. G. G. Cattell. *The Object Database Standard – ODMG-93*. Morgan-Kaufmann Publishers, San Mateo, CA, USA, 1996.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over mulitmedia repositories. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 91–102, Montreal, Canada, June 1996.

[FRV95] D. Florescu, L. Raschid, and P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. In *Proc. CoopIS*, 1995.

[HS93] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–276, Washington, DC, USA, May 1993.

[KHWY96] D. Kossmann, L. Haas, E. Wimmers, and J. Yang. I can do that! using wrapper input for query opitmization in heterogeneous middleware systems, October 1996. Submitted for publication.

[Loh88] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

[LP95] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. In *Proc. ACM Int'l. Conf. on Information and Knowledge Management*, 1995.

[PGH96] Y. Papkonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami, Fl, USA, December 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[Qia96] X. Qian. Query folding. In *Proc. IEEE Conf. on Data Engineering*, pages 48–55, New Orleans, LA, USA, 1996.

[SAC$^+$79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SAD$^+$94] M.-C. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. Pegasus: A heterogeneous information management system. In W. Kim, editor, *Modern Database Systems*, chapter 32. ACM Press (Addison-Wesley publishers), Reading, MA, USA, 1994.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. ICDCS*, 1996.

[TRV97] A. Tomasic, L. Raschid, and P. Valduriez. A data model and query processing techniques for scaling acccess to distributed heterogeneous databases in DISCO. *ACM Trans. Comp. Syst.*, 1997. To appear.

# Optimizing Queries over Multimedia Repositories [†]

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Luis Gravano
Stanford University
gravano@cs.stanford.edu

**Abstract**

*Multimedia repositories and applications that retrieve multimedia information are becoming increasingly popular. In this paper, we study the problem of selecting objects from multimedia repositories, and show how this problem relates to the processing and optimization of selection queries in other contexts, e.g., when some of the selection conditions are expensive user-defined predicates. We find that the problem has unique characteristics that lead to interesting new research questions and results. This article presents an overview of the results in [1]. An expanded version of that paper is in preparation [2].*

## 1 Query Model

In this section we first describe the model that we use for querying multimedia repositories. Then, we briefly review related models for querying text and image repositories.

### 1.1 Our Query Model

In our model, a multimedia repository consists of a set of multimedia objects, each with a distinct object identity. Each multimedia object has a set of attributes, like the date the multimedia object was authored, a free-text description of the object, and the color histogram of an image contained in the object, for example. To query such a repository, users specify a Boolean condition that all objects in the query result should satisfy (the *filter condition*), together with some expression that is used to rank these objects (the *ranking expression*).

The *filter condition* of a query consists of a set of Boolean *atomic conditions* connected together by ANDs and ORs. An atomic condition compares an attribute value of a multimedia object (e.g., the color histogram associated with an image in the object) with a given constant attribute value (e.g., a given color histogram). However, such comparison differs from evaluation of a condition in a traditional selection expression in two ways. First, the comparison is a type-specific method. Unlike in a relational system, where the comparison between two values of the same built-in types is an inexpensive simple predicate, the comparison of two multimedia values can now be *expensive* (e.g., when comparing two color histograms). Second, rarely does a user expect a multimedia object to match a given attribute value exactly. Rather, users are interested in the *grade of match* with which objects match the given attribute values [3]. Thus, given an object $o$, an attribute *attr*, and a constant *value*, the associated grade

[†]Work done while the authors were at Hewlett-Packard Laboratories, Palo Alto, CA.

of match *Grade(attr, value)(o)* is a real number between 0 and 1, and expresses how well $o.attr$ matches $value$. Then, an atomic filter condition is not necessarily an exact equality condition between two values (e.g., between a given color histogram $h$ and the color histogram *oid.color_hist* of an object), but instead an inequality involving the grade of match between the two values together with some target grade. For example, *Grade(color_hist, h)(oid)* $> 0.7$ is an atomic filter condition that is satisfied by all objects whose color histogram matches the given color histogram $h$ with a grade of match higher than 0.7.

Traditional selection queries ask for *all* tuples that match the selection condition, perhaps ordered using the values of a column, or a user-defined function [4]. However, the process of querying and browsing over a multimedia repository is likely to be interactive, and users will tend to ask for only "a few best matches" according to a ranking criterion. Therefore, a query in our model contains a *ranking expression* [3, 5] in addition to the filter condition that we described above. The ranking expression of a query assigns an order to each object in the repository that satisfies the filter condition in the query. In particular, the ranking expression can be a comparison that assigns a grade to each object. These grades are used to sort the objects, so that users can ask for the top 10 objects in the repository by a rank, for example. Ranking expressions can be atomic (e.g., *Grade(color_hist, h)(oid)*), or complex. Complex ranking expressions are built from atomic ones by using the *Min* and *Max* composition functions.

Putting everything together, we use the following SQL-like syntax to describe the queries in our model:

```
SELECT oid
FROM Repository
WHERE Filter_condition
ORDER [k] by Ranking_expression
```

Such a query asks for $k$ objects in the object repository with the highest grade for the ranking expression, among those objects that satisfy the filter condition. The filter condition eliminates unacceptable matches, while the ranking expression orders the acceptable objects.

**Example 1:** Consider a multimedia repository of information on criminals. A record on every person on file consists of a textual description `p` (for profile), a scanned fingerprint `f`, and a recording of a voice sample `v`. Given a target fingerprint `F` and voice sample `V`, the following example asks for records (1) whose fingerprint matches `F` with grade 0.9 or higher, or (2) whose profile matches the string `'on parole'` with grade 0.9 or higher and whose voice sample matches `V` with grade 0.5 or higher. The ranking expression ranks the acceptable records by the maximum of their grade of match for the voice sample `V` and for the fingerprint `F`. The answer contains the top 10 such acceptable records. (For simplicity, we omitted the parameter `oid` in the atomic conditions below.)

```
SELECT oid
FROM   Repository
WHERE  (Grade(v, V) >= .5 AND Grade(p, 'on parole') >= .9)
        OR Grade(f, F) >= .9
ORDER [10] BY Max(Grade(f, F), Grade(v, V))
```

An interesting expressivity question is whether we actually need both the filter condition $F$ and the ranking condition $R$. In other words, we would like to know whether we can "embed" the filter condition $F$ in a new ranking expression $R_F$ such that the top objects according to $R_F$ are the top objects for $R$ that satisfy $F$. In [1], we show that the above is not possible. In other words, our query model would be less expressive without filter conditions.

### 1.2 Related Query Models

Relational query models do not support ranking and grades of match in the sense of Section 1.1. On the other hand, extensible architectures of the *universal servers* can be exploited to support our proposed query model to some extent. An example of how to exploit an extensible architecture is Chabot [6], an image server based on Postgres. Chabot indirectly manages non-exact matches of, say, color histograms through a user-defined predicate *MeetsCriteria*. Given a color histogram $h$ and some "criterion" *Mostly Red*, the predicate *MeetsCriteria("Mostly Red", h)* holds if histogram $h$ is "sufficiently red," according to some hard-coded specification. Unfortunately, the Chabot approach is inflexible in that an explicit user-defined predicate would have to be created to handle ranking expressions that involve several attributes (e.g., to combine grades of match involving the image color histogram and the text caption of a multimedia object). However, an interesting open question is to study how and to what extent we can exploit the capabilities of an extensible architecture like that of Postgres to implement our proposed query model.

The models developed by the information retrieval community to query text repositories support ranking extensively [7]. In particular, the *vector-space* retrieval model typically uses lists of words as queries. Given a list of words, each document is assigned a grade of match for the query, which expresses how *similar* the document and the query are. Then, the documents are sorted based on this grade of match. In this model, both documents and queries are viewed as weight vectors, with one weight per word in the vocabulary. The weight associated with a word and a document (or query) is generally determined by the number of times that the word appears in the document and in the repository where the document occurs. The most common way to compute the grade of match of a document and a query is by taking the inner product of their corresponding weight vectors.

## 2 Storage Level Interface

So far we have defined our query model without describing the behavior and interface of the multimedia repositories. A repository has a set of multimedia objects. We assume that each object has an id and a set of attribute values, which we can only access through indexes. Given a value for an attribute, an index supports access to the ids of the objects that match that value with a certain grade, as we discuss below. Indexes also support access to the attribute values of an object given its oid.

Atomic filter conditions evaluate to either true or false, as discussed in the previous section. We assume that repositories support a *search* interface, denoted by *GradeSearch(attribute, value, min_grade)*. This call returns the ids of the objects that match the given value for the specified attribute with grade at least *min_grade*, together with the grades for the objects.

Also, repositories support a *probe* interface, denoted by *Probe(attribute, value, {oid})*. Given a set of object ids and a value for an attribute, this call returns the grade of each of the specified objects for the attribute value.

## 3 Query Processing

In this section we discuss how to process queries in the model defined above. In Section 3.1 we describe the processing of queries consisting of just a filter condition, which has close ties with traditional query processing. Later, we consider queries with both a filter condition and a ranking expression. How do we process such a query, given that we require only the top $k$ matches? Can we use $k$ to form a "pseudo-selection" condition that can be exploited for processing the entire query? This indeed is the case and we describe this novel approach to query processing in Section 3.2.

## 3.1 Processing Filter Expressions

The problem of optimizing queries that consist of just a filter condition (i.e., with no ranking expression) is indeed a "traditional" problem. Past work has addressed this problem only for some important special cases. However, a more general solution is needed in the context of multimedia selection queries.

For simplicity, we begin by considering the execution space and optimization of a *conjunctive* filter condition, i.e., a filter condition that is the conjunction of a set of atomic conditions. One way to determine the objects that satisfy such a filter condition is to scan the list of all object ids and evaluate each atomic condition for each object using the *probe* interface. The best such plan can be determined by viewing the problem as that of ordering a set of expensive selection conditions. The optimal order is the same as the ascending order of the *rank* $\frac{c}{1-s}$ of each condition, where $c$ is the cost of evaluating the atomic condition for one object, and $s$ is the selectivity of the condition [8]. These plans require executing a *Probe* for each condition and for each object that has a chance to be in the final result. Therefore, for a large repository, this step can be prohibitively expensive.

If one or more of the conditions support a *search* interface, we can do an "indexed" search. For example, we can use one condition for *GradeSearch*, and evaluate the rest of the conditions for the objects selected by using *Probe*. The optimization algorithm consists of estimating, for each choice of condition to use for *GradeSearch*, the cost of *GradeSearch* and the subsequent probing costs. The algorithm picks the condition that provides the least overall cost as the condition to be used for *GradeSearch*.

What happens when the filter condition is not a conjunction of atomic conditions, but instead a condition with atomic conditions connected via ORs as well as ANDs? We can view such a filter condition as an *AND-OR tree* where the tree structure reflects the nesting of propositional operators. As in the case of conjunctive queries, we can scan the list of object ids and then evaluate the atomic conditions to determine the answer to the query. Unfortunately, unlike the case of conjunctive queries, the problem of ordering the evaluation of atomic conditions for an arbitrary propositional condition is intractable [9], hence the need to rely on one of the well known heuristics [10] (cf. [11]).

How can we exploit the *search* interface of repositories for processing arbitrary filter conditions? Unlike the case of conjunctive queries, it is not sufficient to use *GradeSearch* on only one atomic condition to avoid "scanning" all the objects. Thus, the problem of identifying the *search-minimal* condition sets arises. We define a set of conditions $S$ in a query to be search minimal if every object that qualifies to be in the answer set must belong to the result of searching on one of the conditions in $S$. In other words, searching on all conditions of $S$ guarantees that we do not need to do a "scan" of all the objects. Furthermore, if any of the conditions in $S$ is removed, then searching on the reduced set is no longer sufficient to guarantee the completeness of the answer. Thus, a *search-minimal execution* of a filter condition $f$ searches the repository using a search-minimal condition set $m$ for $f$, and executes the following steps. (In the following, we assume independence of the atomic conditions in the filter condition.)

- Determine an optimal search-minimal condition set $m$.
- For each condition $a \in m$:
  - Search on $a$ to obtain a set of objects $S_a$.
  - Probe every object in $S_a$ with the residual condition $R(a, f)$ to obtain a filtered set $S'_a$ of objects that satisfy $f$.
- Return the union $\bigcup_{a \in m} S'_a$.

Thus, we have two new challenges here, even in the limited context of the search-minimal execution space: (a) Given a filter condition $f$ and a search condition $a$, what is the residual predicate $R(a, f)$ that an object retrieved via search on $a$ must satisfy to be in the answer set? (b) How do we determine the optimal search-minimal condition set? Our paper [1, 2] provides polynomial time algorithms to answer both questions under broad assumptions of cost models. We illustrate the step of determining residual predicates in the example below. The algorithm to determine an optimal search-minimal condition set for an independent filter condition is in [1]. Intuitively, the algorithm traverses the AND-OR condition tree in a bottom-up fashion. For each AND node, it takes the optimal search-minimal condition set of one of the subtrees (the one with minimal cost), and for each OR node, it takes the union of the search-minimal condition sets of all its subtrees.

**Example 2:** Consider the filter condition $f = a_4 \wedge ((a_1 \wedge a_2) \vee a_3)$. The residue of the atomic condition $a_2$ is $R(a_2, f) = a_1 \wedge a_4$. As another example, $R(a_4, f) = (a_1 \wedge a_2) \vee a_3$. Then, any object that satisfies $a_4$ and also satisfies $R(a_4, f)$ satisfies the entire condition $f$. Observe that each of $\{a_4\}$, $\{a_2, a_3\}$, and $\{a_1, a_3\}$ is a search-minimal condition set. If we decide to search on $\{a_2, a_3\}$, the following three steps yield exactly all of the objects that satisfy $f$:

1. Search on $a_2$ and probe the retrieved objects with residue $R(a_2, f) = a_1 \wedge a_4$. Keep the objects that satisfy $R(a_2, F)$.
2. Search on $a_3$ and probe the retrieved objects with residue $R(a_3, f) = a_4$. Keep the objects that satisfy $R(a_3, F)$.
3. Return the objects kept.

The discussion above can be cast in a more traditional framework. We investigated the problem of finding an optimal execution for a selection in the presence of indexes as well as user-defined methods for arbitrary propositional clauses. We focussed on search-minimal executions, which correspond to using a minimal number of indexes to avoid sequential scans of the data. Thus, when filter conditions are a conjunction of literals, a search-minimal execution corresponds to using a single index. We showed that we needed new algorithms to handle arbitrary filter conditions even in the context of search-minimal executions. Further complexity arises if we step beyond the search-minimal execution space. This corresponds to doing index intersections to evaluate a query (e.g., for processing a conjunctive filter condition). In this case, our problem would become closely related to that of query processing with index AND-ing and OR-ing [12]. However, our model would require that we extend the existing results, since *Probe* costs are not zero anylonger. Furthermore, not only would we need to choose the superset of a search-minimal set to search, but we should also address the more complex ordering of search-result merges and probes [2].

## 3.2 Processing Ranking Expressions

We now look at queries consisting only of ranking expressions. Such queries have the following form:

```
SELECT oid
FROM Repository
ORDER [k] by Ranking_expression
```

The result of this query is a list of $k$ objects in the repository with the highest grade for the given ranking expression. The ranking expressions are built from atomic expressions that are combined using the *Min* and *Max* operators (Section 1.1).

At first glance, processing such queries appears to be troublesome. Although we need no more than $k$ best objects, we must retrieve each object, evaluate the ranking expression over each object, and then sort the objects accordingly. In other words, although we are interested in only the top $k$ objects, we are unable to take advantage of $k$ for query processing. We show that we can map a given ranking expression into a filter condition, and process the ranking expression "almost" as if it were a filter condition. This result makes it possible to take advantage of the parameter $k$ for query processing and is central to processing queries with ranking expressions using the techniques of Section 3.1 for filter conditions.

Fagin presented a novel approach to take advantage of $k$ in processing a query consisting of a ranking expression such as $R = \mathit{Min}(a_1, \ldots, a_n)$, where the $a_i$'s are independent atomic expressions [3]. Fagin has proved the important result that his algorithm to retrieve $k$ top objects for an expression $R$ that is a *Min* of independent atomic expressions is asymptotically optimal with arbitrarily high probability in terms of the number of objects retrieved.

Although Fagin's strategy helps reduce the number of objects that need to be retrieved to process a ranking expression, it cannot treat a ranking expression as a selection condition from the point of view of query processing (e.g., to determine the search-minimal condition set). Our key contribution is not only to take advantage of $k$ (which Fagin did) but also to view a ranking expression as a filter expression so as to make query processing and cost-based optimization of queries uniform irrespective of the presence of ranking expressions.

When the query contains both a filter condition $F$ and a ranking expression $R$, it asks for $k$ top objects by the ranking expression $R$ that satisfy $F$. Using the results in this section, we can translate this query into the problem of optimizing the filter condition $F \wedge F'$, where $F'$ is the filter condition "associated" with $R$. We now describe how such an $F'$ is determined.

Given a ranking expression $R$ and the number $k$ of objects desired, we show that:

1. There is an algorithm to assign a grade to each atomic expression in $R$, and a filter condition $F$ with the same "structure" as $R$, such that $F$ is expected to retrieve at least the top $k$ objects according to $R$.

2. There is a search-minimal execution for $F$ that retrieves an expected number of objects that is no larger than the expected number of objects that Fagin's algorithm would retrieve for $R$ and $k$.

**Example 3:** Consider a ranking expression $e = \mathit{Min}(\mathit{Grade}(a_1, v_1), \mathit{Grade}(a_2, v_2))$, where $a_i$ is an attribute, and $v_i$ a constant value. We want two objects with the top grades for $e$. Now, suppose that we can somehow find a grade $G$ (the higher the better) such that there are at least two objects with grade $G$ or higher for expression $e$. Therefore, if we retrieve all of the objects with grade $G$ or higher for $e$, we can simply order them according to their grades, and return the top two as the result to the query.

In other words, we can process $e$ by processing the following associated *filter condition* $f$, followed by a sorting step of the answer set for $f$:

$$f = (\mathit{Grade}(a_1, v_1)(o) \geq G) \wedge (\mathit{Grade}(a_2, v_2)(o) \geq G)$$

By processing $f$ using the strategies in Section 3.1, we obtain all of the objects with grade $G$ or higher for $a_1$ and $v_1$, and for $a_2$ and $v_2$. Therefore, we obtain all of the objects with grade $G$ or higher for the ranking expression $e$. If there are enough objects in this set (i.e., if there are at least two objects), then we know we have retrieved the top objects that we need to answer the query with ranking expression $e$. Similarly, we can process a ranking expression $e' = \mathit{Max}(\mathit{Grade}(a_1, v_1), \mathit{Grade}(a_2, v_2))$ as filter condition $f' = (\mathit{Grade}(a_1, v_1)(o) \geq G') \vee (\mathit{Grade}(a_2, v_2)(o) \geq G')$, for some grade $G'$.

The example above shows how we can process a ranking expression $e$ as a filter condition $f$ followed by a sorting step. But the key point in mapping the ranking problem to a (modified) filtering problem is finding the grade $G$ to use in $f$. In [1], we present the algorithm `Grade_Rank`, which given the number of objects desired $k$, a ranking expression $e$, and selectivity statistics, produces the grade $G$ for the filter condition $f$.

Our approach allows us to translate the ranking expressions into filter conditions, and to use the processing strategy of Section 3.1. At least $k$ objects are expected to satisfy $F$. However, if at run time we find that fewer than $k$ objects satisfy $F$, we should lower the grade $G$ used in $F$. We will investigate strategies to lower $G$ as part of our future work.

It is natural to ask how our algorithm compares with Fagin's in terms of retrieval efficiency. In [1], we show that if we process a ranking expression (and its associated number $k$ of objects requested) by using a filter condition $F$ with grade $G$ as determined by algorithm `Grade_Rank`, we can expect to retrieve no more objects than Fagin's algorithm, under some assumptions on the repositories. Furthermore, our experiments show that the approach that we outlined in this section still is a desirable one when the assumptions on the repository do not hold strictly.

Although in this section we showed how to process a ranking expression like a filter condition, the semantics of both the filter condition and the ranking expression remain distinct. (See Section 1.1.) After processing a ranking expression as a filter condition, we have to compute the grade of the retrieved objects for the ranking expression, and sort them before returning them as the answer to the query.

### 3.3 Putting our Results in Perspective

Consider queries that do not have a ranking expression. Such queries consist of a filter condition with AND and OR Boolean connectives where some of the atomic conditions can be expensive. Our results on processing such queries generalize past work on processing conditions with expensive predicates. In effect, our algorithms consider the case where both of the following conditions are true: (a) the filter condition is more general than a conjunction of literals, and (b) an expensive predicate can be evaluated by using *GradeSearch* (indexed access) as well as by using *Probe* (both the arguments of the predicate are bound). Although past work on expensive predicates addresses the case where only condition (b) holds [13, 14], it does not address the case where condition (a) holds as well. Finally, note that the algorithms and results of the previous section are completely *independent* of the nature of the atomic predicates as long as a selectivity and a cost measure are available.

Several interesting query processing questions remain open. The search-minimal execution space is restrictive in a way analogous to using only one index for processing a traditional selection condition. Eliminating the restriction that executions be search minimal opens interesting directions to explore.

Our ranking expressions are built using the *Min* and *Max* operators. Another interesting question to explore is how to process ranking expressions that use operators like a weighted sum, for example, and deciding what other operators would be useful to allow in ranking expressions.

## References

[1] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. In *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, pages 91–102, Montreal, June 1996.

[2] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. Technical report, 1996. In preparation.

[3] Ronald Fagin. Combining fuzzy information from multiple systems. In $15^{th}$ *ACM Symposium on Principles of Database Systems*, June 1996. Also available as IBM Almaden Research Center Technical Report RJ 9980.

[4] Jim Gray et al. Data cube: A relation aggregation operator generalizing group by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1996.

[5] F. Rabitti. Retrieval of multimedia documents by imprecise query specification. In *Proceedings of the 1990 EDBT*, Venice, March 1990.

[6] Virginia E. Ogle and Michael Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9), September 1995.

[7] Gerard Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Addison Wesley, 1989.

[8] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, Washington D. C., May 1993.

[9] L. T. Reinwald and R. M. Soland. Conversion of limited entry decision tables to optimal computer programs I: Minimum average processing time. *Journal of the ACM*, 13(3), 1966.

[10] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object bases. In *Proceedings of the 18th VLDB Conference*, Vancouver, August 1992.

[11] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, Minneapolis, May 1994.

[12] C. Mohan. Single table access using multiple indexes: Optimization, execution and concurrency control techniques. In *Proceedings of the 1990 EDBT*, Venice, March 1990. Also available as IBM Almaden Research Center Technical Report RJ 7341.

[13] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. In *Proceedings of the 22nd VLDB Conference*, Bombay, India, September 1996.

[14] J. Hellerstein. Optimization techniques for queries with expensive methods. Technical Report CS-TR-96-1304, University of Wisconsin, Madison, February 1996. Ph.D. dissertation.

**CALL FOR PARTICIPATION** ICDE 13

13th International Conference on

# Data Engineering

April 7 - 11, 1997

University of Birmingham, Birmingham, UK

**Sponsored by the IEEE Computer Society**

DATA ENGINEERING

IEEE COMPUTER SOCIETY 50 YEARS OF SERVICE • 1946-1996

Data Engineering deals with the use of engineering disciplines in the design, development and evaluation of information systems for different computing platforms and architectures. The 13th Data Engineering Conference will provide a forum for the sharing of original research results and practical engineering experiences among researchers and practitioners interested in all aspects of data engineering and knowledge management. The purpose of the conference is to share solutions to problems that face our information-oriented society and to identify new challenges and directions for future research.

## Current Information

Up-to-date information on the 13th International Conference on Data Engineering, including advance program, conference registration, and hotel information and reservations can be found at the ICDE-97 conference WWW sites:

**USA WWW Site:**
http://www.cse.ogi.edu/icde97/

**European WWW Site:**
http://www.scit.wlv.ac.uk/icde97/

## Organizing Committee

**General Co-Chairs:**
Mike Jackson
University of Wolverhampton, UK

Calton Pu
Oregon Graduate Institute, USA

**Program Co-Chairs:**
Alex Gray
University of Wales, Cardiff, UK

Paul Larson
Microsoft Corporation, USA

**Industrial Program Chair:**
Jose Blakely
Microsoft Corporation, USA

**Panel Program Chair:**
Wolfgang Klas
University of Ulm, Germany

**Tutorial Program Co-Chairs:**
Ming-Syan Chen, National Taiwan University, Taiwan, ROC

Carole Goble
University of Manchester, UK

## Invited Speakers

Invited presentations by Bill Baker (Microsoft): "*Universal Access Versus Universal Storage*" and Malcolm Atkinson (University of Glasgow): "*Java and Databases: persistence, orthogonality, and independence*".

## Research Program Sessions

Object Schemas and Versions

Mobile Computing

Managing Video Data

OLAP and Data Mining

Data Warehousing and Data Mining

Indexing and Prefetching in OOBDMS

Transaction Management

Database Interoperability

Query Processing

Performance Issues

Spatial Databases and GIS

Scalability in Distributed Systems

Real-time Databases and Concurrency

Data Modeling

Dynamic Roles and Moving Objects

## Industrial Program Sessions

Multimedia Application Requirements

Data Warehousing, client side

Data Warehousing, server side

New Hardware Platforms (clusters)

## Panels

This year's conference features a number of panels featuring lively discussion and debate by technical innovators. Panels discussions include: "*Database Research: lead, follow, or get out of the way?*", "*Nomadic and Mobile Computing*", "*OO or Object-Relational: which way to go?*", "*Data Mining: where is it heading*", "*Databases and the Web: what s in it for the databases?*", and "*Temporal Databases: is TSQL a mistake?*".

## Tutorials

The conference also features 6 half-day tutorials, held April 7 and April 8.

"*CORBA and Distributed Object Technologies,* Sean Baker, Iona Technologies, Eire. Monday morning, April 7.

"*Work ow Management Systems: Programming, Architecture, Commercial Products,*" Berthold Reinwald, IBM Almaden. Monday afternoon, April 7.

"*Intelligent Agents,*" Michael Wooldridge, Mitsubishi, UK and Nicholas Jennings, Queen Mary & West□eld College, London UK. Monday afternoon, April 7.

"*Databases and the World Wide Web,*" Marianne Winslett, University of Illinois, USA. Tuesday morning, April 8.

"*Geographical Information Systems,*" Hanan Samet, University of Maryland, USA. Tuesday afternoon, April 8.

"*Integration of Data Mining and Data Warehousing Technologies,*" Jiawei Han, University of British Columbia, Canada. Tuesday afternoon, April 8.

**Conference WWW Pages -USA Site**: http://www.cse.ogi.edu/icde97/ **European Site**: http://www.scit.wlv.ac.uk/icde97/