SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:355–384
Published online 27 August 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.898

Seamlessly integrating similarity queries in SQL

M. C. N. Barioni[1,*,†], H. L. Razente[2], A. J. M. Traina[2] and C. Traina Jr[2]

[1]*Centro de Matemática, Computação e Cognição, Universidade Federal do ABC (UFABC), Rua Santa Adélia, 166, Bairro Bangu, 09210-170, Santo André, SP, Brazil*
[2]*Departamento de Ciências de Computação, Universidade de São Paulo, Campus de São Carlos, Caixa Postal 668, 13560-970 São Carlos, SP, Brazil*

SUMMARY

Modern database applications are increasingly employing database management systems (DBMS) to store multimedia and other complex data. To adequately support the queries required to retrieve these kinds of data, the DBMS need to answer similarity queries. However, the standard structured query language (SQL) does not provide effective support for such queries. This paper proposes an extension to SQL that seamlessly integrates syntactical constructions to express similarity predicates to the existing SQL syntax and describes the implementation of a similarity retrieval engine that allows posing similarity queries using the language extension in a relational DBMS. The engine allows the evaluation of every aspect of the proposed extension, including the data definition language and data manipulation language statements, and employs metric access methods to accelerate the queries. Copyright © 2008 John Wiley & Sons, Ltd.

Received 17 July 2007; Revised 22 February 2008; Accepted 1 July 2008

KEY WORDS: similarity queries; content-based retrieval; metric access methods; SQL extension

INTRODUCTION

Contemporary database management systems (DBMS) must deal with the increasingly diversified data being generated. Complex data types such as multimedia data (image, audio, video and long text), geo-referenced information, time series, fingerprints, genomic data and protein sequences,

*Correspondence to: M. C. N. Barioni, Universidade Federal do ABC (UFABC), Rua Santa Adélia, 166, Bairro Bangu, 09210-170, Santo André, SP, Brazil.
†E-mail: camila.barioni@ufabc.edu.br

Contract/grant sponsor: CNPq (Brazilian National Council for Supporting Research)
Contract/grant sponsor: CAPES (Brazilian Coordination for Improvement of Higher Level Personnel)
Contract/grant sponsor: FAPESP (São Paulo State Research Foundation)

among others, need to be stored in DBMS and retrieved through meaningful and powerful commands of the standard query language, the SQL (Structured Query Language).

The searching operations performed by traditional DBMS rely on the total ordering property held by numbers and short-texts, which enable their comparison using the relational operators $<$, $\leq$, $>$ and $\geq$. Unfortunately, the majority of complex data domains do not have the total ordering property, precluding the use of the relational operators to compare them. The exact match operators ($=$ or $\neq$) are also not meaningful, as two identical elements (e.g. two identical images) rarely occur. Fortunately, complex data domains often allow the definition of similarity relationships among pairs of objects, which enables them to be queried by similarity. As similarity-based comparisons require a way to quantify how similar each pair of objects is, complex data domains are often considered as metric spaces [1].

Formally, a metric space is a pair $\langle \mathbb{S}, d() \rangle$, where $\mathbb{S}$ is the set of all objects complying with the properties of the domain and $d()$ is a distance function that complies with the following three properties: *symmetry*: $d(s_1, s_2) = d(s_2, s_1)$; *non-negativity*: $0 < d(s_1, s_2) < \infty$ if $s_1 \neq s_2$ and $d(s_1, s_1) = 0$; and *triangular inequality*: $d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2)$, $\forall s_1, s_2, s_3 \in \mathbb{S}$. Vector data sets with any L_p distance function, such as the Euclidean distance (L_2), are special cases of metric spaces. The similarity of two objects can be calculated by appropriately defining a distance function to create a metric space.

Some domains of complex objects are seldom directly compared. Rather, it is usual to extract features from each object and compare their features instead of the objects themselves. Often complex objects can be compared in different ways. For example, images can be compared by the similarity of their color distribution or by the similarity of their texture. This can be accomplished by extracting several features that are stored together with the object itself and by defining a distance function specific for each way in which the objects can be compared.

As objects of complex types are being stored at an increasing rate in relational databases, the need to support similarity queries also increases. Several works have been published reporting basic algorithms to execute similarity retrieval operations on data sets of complex objects [2–4]. However, there are few works on languages to represent similarity queries, and the current standard of SQL does not consider them either. Although we started to address this issue in our previous papers [5,6], there are more details to be carried out herein. In [6] we presented the concepts regarding similarity queries for the manipulation of complex objects represented by a set of traditional attributes and in [5] we described a demonstration of a similarity retrieval engine (SIREN) applied to some example applications together with a brief presentation of its architecture. This paper integrates the concepts presented in our previous papers and extends them aiming to provide the support for similarity queries considering another representation of complex objects, such as the ones that depend on feature extraction algorithms, a detailed description of the architecture and functionalities of the SIREN, and also, new similarity predicates that increase the range of applications that can benefit from the use of similarity queries. Thus, this paper aims at providing support for similarity queries in SQL, meeting the following requirements:

- allowing a flexible representation of powerful similarity queries over complex contents, yet having a low impact over the existing SQL syntax;
- supporting similarity queries over objects represented explicitly, e.g. as a BLOB, and over objects represented by a set of traditional attributes, using the same syntactical constructions;

DOI: 10.1002/spe

- enabling the optimization of similarity queries, following the successful optimization processing of relational queries expressed in SQL;
- supporting any number of user-defined feature extractors and adjustable distance functions over complex data, possibly granting them to every attribute defined in the same data domain.

Seamlessly integrating similarity queries with the existing queries already supported by SQL is important to allow optimizing the full set of search operations involved in each query. Today, several highly optimized algorithms exist for each specific similarity operation. A procedural approach to develop the extension, such as employing user-defined types (UDT) and user-defined functions (UDF) from object-relational databases, can take advantage of them. However, a procedural approach does not allow exploiting the optimization opportunities provided by the combination of multiple similarity operations or by the integration of similarity and the other query operations that can occur in a single query. In fact, relying on UDF to support a large class of queries, as the similarity queries, removes the relational paradigm from the language and moves it to the imperative realm, thus denying many optimization opportunities that could be otherwise exploited. Therefore, integrating similarity queries in a fully relational approach, as proposed in this paper, is a fundamental step to allow supporting complex objects as 'first class citizens' in modern DBMs, as the traditional data types are.

In the following sections, we summarize existing related work and present concepts regarding types of similarity queries used in this work. In addition, we present the new syntax proposed to add similarity queries in SQL and discuss how similarity queries can be expressed and executed and how complex data can be stored to allow answering similarity queries. Finally, at the end we give the conclusions of this paper and suggestions for future works.

RELATED WORK

Querying by similarity is a fundamental paradigm when searching large collections of complex data. Many works aim at increasing the efficiency of similarity query algorithms, such as index structures tailored to answer them [7,8]. On the other hand, few works aiming at extending SQL to support similarity queries have been presented. Simple SQL extensions were proposed in [9,10]. Although these works had pursued ways to support similarity queries in SQL, they are not able to provide a production-strength support seamlessly integrated with the other features of SQL. Moreover, they do not deal with all the predicates needed to process similarity queries or with its integration with non-similarity-based predicates into the same query command.

The International Standards Organization is working on the SQL/MM (SQL Multimedia and Application Packages, ISO/IEC 13249) [11], a multi-part standard proposal that provides storage and manipulation support for several multimedia data types based on UDT and UDF. However, the standard proposal does not cover how to query these data types, especially considering the requirements of content-based retrieval. This proposal already provides specific data types for the storage of complex data and allows the user to specify operations based on ranking functions[‡],

[‡]A ranking function returns a ranking value for each row in a relation.

DOI: 10.1002/spe

one for each search type required. Thus, each similarity-based predicate in a query corresponds to a UDF call. The following command illustrates a query involving traditional and non-traditional predicates:

```
SELECT *
  FROM <table_names>
 WHERE <similarity-based_predicate1>
   AND <similarity-based_predicate2>
   AND <traditional_predicate>;
```

Although the UDF/UDT approach can use the existing highly optimized algorithms to process each specific similarity predicate, it allows neither optimizations among several predicates nor their integration with other non-similarity-based predicates that can compose a query. In a query involving several similarity-based predicates, as the one illustrated above, each predicate is considered as an isolated operation; thus, it does not take advantage of optimizations that can be performed combining simpler expressions. Therefore, queries composed of multiple similarity-based predicates require expensive union and intersection operations to combine intermediate results to answer conjunctions and disjunctions of elementary predicates. Moreover, the optimization for queries that employ similarity-based predicates defined as UDF cannot be properly optimized by DBMS, as user-defined functions evaluate each row as soon as possible, and no *a priori* knowledge is assumed with respect to each UDF.

As a consequence, there is a need for an approach that enables the use of similarity operators together with other predicates already available in SQL, such as exact match and order-based comparisons for textual/numeric attributes, allowing optimization opportunities. This is the aim of the work presented herein. It is important to note that, although our approach allows optimization opportunities, we do not address this aspect in this paper. As a matter of fact, an optimizer produces only alternative equivalent plans to find a better way to execute similarity queries. In order to allow choosing among equivalent plans, the similarity-based predicates should be included into the relational algebra. These are subjects of current research projects and have been addressed in [12,13], which are complementary to the present work.

Commercial products such as Oracle InterMedia and IBM DB2 IAV Extenders follow the SQL/MM approach. Table I compares the main aspects of our proposed approach with the representative available systems that allow posing similarity queries.

FUNDAMENTAL CONCEPTS

Complex objects are stored in a database either as a set of traditional attributes or as one attribute represented as a large binary object (BLOB). For example, geo-referenced information and time series are stored as sets of real-valued attributes, whereas images and audio tracks are usually stored as BLOB data. In this paper, the former are called *particulate* objects and the latter are called *monolithic* objects. Particulate objects can be compared by similarity using a distance function defined over their constituting attributes. For example, geo-referenced objects usually employ the Euclidean distance function over the coordinates of the object. Monolithic objects require extracting predefined features that are used in place of each object to define the distance function. For example,

DOI: 10.1002/spe

Table I. Comparison of similarity query approaches.

Feature	Our approach	Oracle*	DB2†
Representation of similarity queries	Native predicates	Ranking functions	Ranking functions
Optimizations among similarity operators and/or relational operators	Yes	No	No
Multiple distance functions when defining similarity measures	Yes	No	No
Inclusion of new extractors	Yes	Yes	Yes

*Oracle Intermedia 10g Release 2.
†DB2 AIV Extenders Version 8.

images are preprocessed by specific feature extraction algorithms to retrieve their color and texture histograms, polygonal contours of the pictured objects, etc., which are used to define the corresponding distance functions. Querying monolithic objects is commonly called retrieval by content (as in content-based image retrieval (CBIR) for images) [14].

Admittedly, it is conceivable that complex domains with particular properties should be fully supported by a single syntax covering all complex domains. Therefore, in this paper we present the support to particulate domains and the image and audio types as representatives of the monolithic domains, although many other monolithic domains could be equally supported.

Considering relational DBMS, the elements of a given domain can be compared using any property defined over the domain, which includes every pair of objects from one attribute and from attributes sharing the same domain. Regarding complex objects, the elements must be compared by similarity using a distance function defined over the domain. A similarity query is expressed through similarity predicates, and each predicate involves attributes of complex data types and a distance function $d()$. An attribute of a complex data type is an object whose value is an element s_i from the corresponding complex domain $\mathbb{S}$. The set of values of the complex objects stored in the database forms the data set S. In a similarity query, the distance function ranks the objects in S up to a given limit *lim*. There are basically two ways to limit the number of returned objects. The first is based on a given similarity threshold ξ, and the second is based on a number k of objects.

The distance function $d()$ compares pairs of objects from one complex domain $\mathbb{S}$, assigning to each pair a non-negative real value, that is, $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}$. One of the operands always comes from one data set $S \in \mathbb{S}$. The other operand can be either a set of objects given as part of the predicate or a set of objects from another data set also getting its values from the same domain $\mathbb{S}$. The former are called unary predicates and the latter are called binary predicates.

Below we list the most important similarity predicates found in the literature. They are presented in a way that enables them to be integrated with the existing non-similarity-based predicates already supported by SQL and with the relational algebra. To illustrate the similarity predicates, we use the following relation that stores data from employees:

```
Employee={Name, HomeCoord, Mugshot}
```


DOI: 10.1002/spe

where `Name` is a string attribute, `HomeCoord` is a complex object storing the coordinates of the employee's home and `Mugshot` is a complex object storing the photo of the employee.

Unary predicates

Unary similarity predicates correspond to similarity selections and can be expressed through the use of similarity operators. They compare the elements of one data set $S \subset \mathbb{S}$ with one or more constant values s_q, called reference elements, taken from the domain $\mathbb{S}$ and given as part of the predicate. The set of reference elements s_q, called the query data set, is denoted as Q. The answer to a similarity selection is a subset of objects $s_i \in S$ that meets the query predicate.

Single center unary predicates

When the query data set Q has only one element s_q, the commonest cases of the similarity selections are the range and k-nearest-neighbor queries:

- *Range selection—Rq*: Given a maximum query distance ξ as the limit *lim*, the $\hat{\sigma}_{(Rq(s_q,\xi))}S$ query retrieves every object $s_i \in S$ that satisfies $d(s_i, s_q) \leq \xi$. An example is: 'select the employees whose home's coordinates are distant from position $\langle x, y \rangle$ by at most 5 km, considering the Euclidean distance $L_2()$'.

  Figure 1(a) shows an example of a range selection in an objects set considering different distance functions L_p for a maximum query distance ξ.
- *k-nearest neighbor selection—kNN*: Given an integer value $k \geq 1$, the $\hat{\sigma}_{(kNN(s_q,k))}S$ query retrieves the k objects $s_i \in S$ that have the smallest distances from the query object s_q, according to the distance function $d()$. An example is: 'Select the 3 mugshots most similar to mugshot Im regarding texture'.

  An example of a k-nearest-neighbor selection, with $k = 4$, is shown in Figure 1(b).

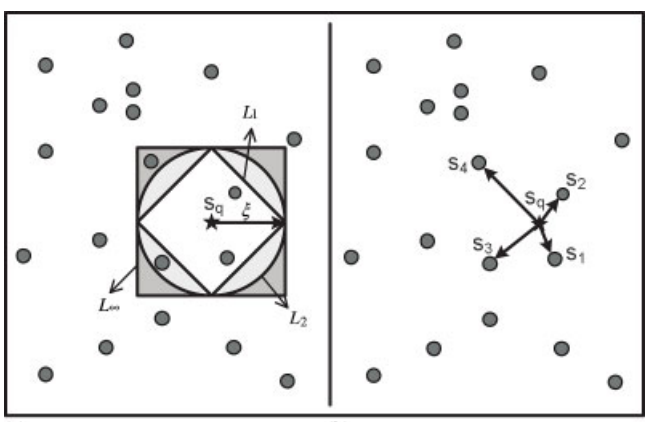

Figure 1. Examples of similarity selections in an objects set: (a) range selection considering the distance functions L_1, L_2 and L_∞ and (b) k-nearest-neighbor selection considering $k = 4$ and the Euclidean distance.

DOI: 10.1002/spe

Multi-center unary predicates

When the query data set Q has more than one object, the distances from each center $s_q \in Q$ to an object $s_i \in S$ must be aggregated to give the measure of similarity d_g from every object s_i to the set of centers in Q [15]. This measure is used by the similarity operator to rank the objects in S. There are many ways to generate the aggregation. In this paper we consider that it is generated by the summation

$$d_g = \sqrt[p]{\sum_{s_q \in Q} d(s_q, s_i)^g} \tag{1}$$

Different values for the power g can lead to different interpretations. For example, if $g=1$, then the resulting query will rank the objects s_i to minimize the summation of distances to the query centers, and if $g=2$, then the resultant query corresponds to a linear regression returning the objects that minimize the sum of the squared distances from each object to the query centers. Conceptually, this aggregate function considers those objects that would minimize the distances for all objects in Q to evaluate the answer.

There are several applications that can benefit from the use of queries that employ this type of similarity predicates. For instance, if we consider the following situation: a couple wants to decide on which kindergarten to leave their child in, considering three locations: their home, his work and her work. This couple could want to achieve the best answer in terms of a certain criterion, such as the minimization of the total distance that should be traversed, the minimization of the distances from all the destinations or the minimization of the maximum distances that should be traversed.

Besides being able to answer questions such as those posed in the previous example, queries that employ multi-center unary predicates can also play an important role in relevance feedback approaches based on multiple query centers for CBIR [16]. Figure 2 illustrates the three different values for the power g, considering a k-nearest-neighbor selection with $k=1$, three query objects and the Euclidean distance. In this figure, the circles represent the objects of a data set S, the stars

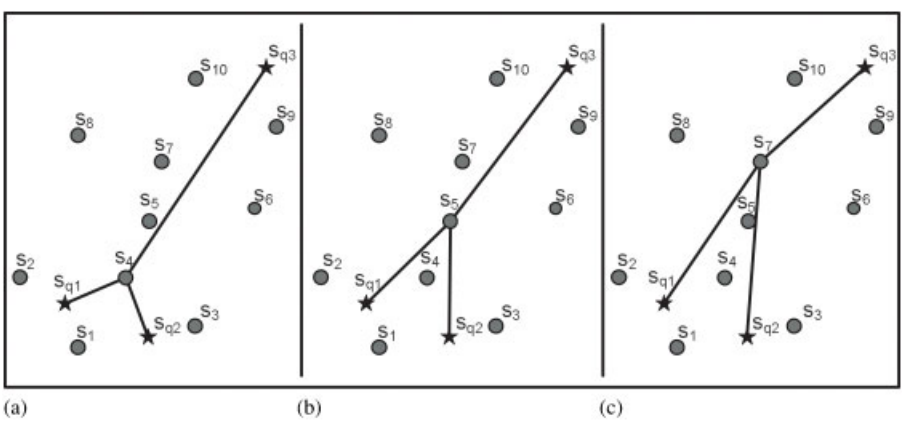

Figure 2. Examples of similarity queries with multi-center unary predicates. Illustration of the use of Equation (1) considering the Euclidean distance and the powers $g=1$ (a), $g=2$ (b) and $g=\infty$ (c).

DOI: 10.1002/spe

represent the query centers of Q and the object attached to the query centers corresponds to the query answer.

Binary predicates

Binary similarity predicates correspond to similarity joins and can also be expressed through the use of similarity operators. They compare the elements of two data sets S, $R \subset \mathbb{S}$ and return pairs of objects $\langle s_i, r_l \rangle | s_i \in S, r_l \in R$ that met the query predicate. Similarity joins can be limited either by a distance ξ or by the number of objects k, as follows:

- *Range join*— $\overset{Rq}{\bowtie}$: Given a maximum query distance ξ, the query $S \overset{Rq[d(),\xi]}{\bowtie} R$ retrieves the pairs of objects $\langle s_i, r_l \rangle | s_i \in S, r_l \in R$ that satisfy $d(s_i, r_l) \leq \xi$. An example is: 'Select the employees of the sales department whose home's coordinates are distant from the employees of the purchase department by at most 5 km, considering the Euclidean distance $L_2()$'. Consider the data sets of employees from the sales and the purchase departments have the same structure of the `Employee` relation.
- *k-nearest-neighbors join*— $\overset{kNN}{\bowtie}$: Given an integer value $k \geq 1$, the query $S \overset{kNN[d(),k]}{\bowtie} R$ retrieves the pairs of objects $\langle s_i, r_l \rangle | s_i \in S, r_l \in R$ such that there are k pairs for each object of S together with its nearest objects from R. Note that, in the answer set, there are k pairs of objects for each different value of $s_i \in S$. An example is: 'Select the 10 mugshots of the purchase department employees that are the most similar to each mugshot of the sales department employees regarding texture'.
- *k-closest-neighbors join*— $\overset{kCN}{\bowtie}$: Given an integer value $k \geq 1$, the query $S \overset{kCN[d(),k]}{\bowtie} R$ retrieves the k closest pairs of objects $\langle s_i, r_l \rangle | s_i \in S, r_l \in R$. Note that there are at most k pairs of objects in the answer set. An example is: 'Select the 20 most similar pairs of mugshots of the sales and purchase department employees regarding texture'.

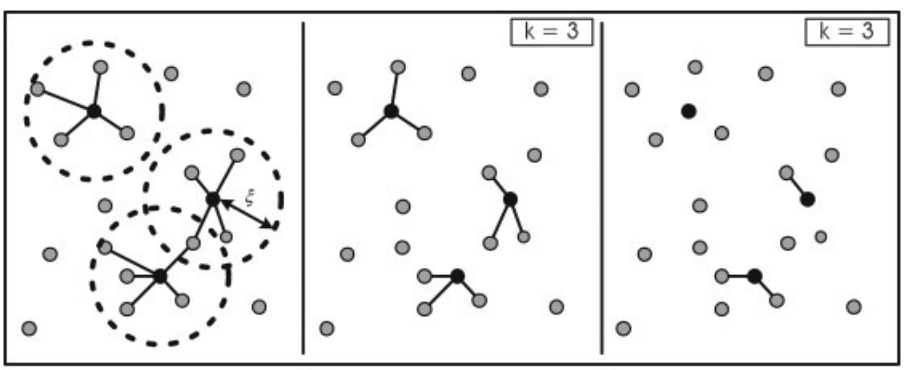

Figure 3. Illustration of the differences among the three types of similarity joins: (a) range join; (b) k-nearest-neighbors join with $k = 3$; and (c) k-closest-neighbors join with $k = 3$.

DOI: 10.1002/spe

Figure 3 shows an illustration of the three types of similarity joins described previously. In this figure, the black circles represent objects of the data set S and the gray circles represent objects of the data set R.

Every similarity operator allows a number of variations, such as retrieving the most dissimilar elements instead of the most similar elements, and taking into account occurrences of ties in k-limited predicates.

THE SIMILARITY RETRIEVAL ENGINE

The SIREN [5,17] was implemented in order to evaluate the adequacy of the language proposed in the 'Supporting Similarity Queries in SQL' section, which allows expressing similarity queries in SQL [6]. SIREN is composed of three main components:

- the interpreter of the extended SQL syntax proposed in this paper to add similarity queries in SQL;
- a set of feature extraction algorithms, which are employed to extract the features used to represent and to compare complex objects;
- the indexer, which deals with the utilization of appropriate index structures developed to answer similarity queries (usually called metric access methods (MAMs)).

The main aspects related to each of these components are presented in the following sections.

The command interpreter

The command interpreter is the main component of SIREN. It performs the lexical and syntactical analysis to determine: which are the complex data and how they must be queried or stored in the application databases and/or in the data dictionary; when and which feature extractors should be used; when and how the complex attributes should be indexed by a MAM; which and how the similarity operations should be performed. SIREN acts like a blade between a conventional DBMS and the application programs. It intercepts and analyzes every SQL command sent from the application. If the command has neither similarity construction nor a reference to complex objects, it sends the command to the underlying DBMS and relays the answer from the DBMS back to the application program. Therefore, when only conventional SQL commands are posed by the application, SIREN is transparent (apart from a slight delay while the command is interpreted). When the SQL command has similarity-related demands or references to complex objects, the command is re-written to internally execute the similarity-related operations, using the underlying DBMS to execute the conventional operations.

Supporting similarity queries in SQL

As stated in the 'Introduction', the first requirement that must be addressed to introduce similarity queries into SQL is: '*How to represent similarity queries over multimedia domains yet having a low impact over the existing SQL syntax*?' To fulfill it, each domain (particulate or monolithic) where the similarity must be measured needs to be defined as a new data type. In this paper we restrict the discussion regarding monolithic domains to audio and images, exemplifying the concepts by

DOI: 10.1002/spe

creating in SQL the new data types AUDIO and STILLIMAGE and all the tools required to process them. However, other types of monolithic domains, such as video and time sequences, can be manipulated by following the same approach.

To actually execute a similarity query over complex objects, it is needed both to be able to represent the similarity query and to express how to compare pairs of complex objects. In the following discussion, we first assume that 'how to compare complex objects' was previously defined and present how to pose similarity queries, showing that the proposed query constructs are in fact independent from the target object type and from the way in which they can be compared. Thereafter we present how to compare complex objects, in a way that the similarity queries can be answered.

Specifying similarity queries in the SELECT command. The following description of the proposed extension of the data manipulation language (DML) commands from SQL assumes that the complex attributes where previously defined in the existing relations, as well as their corresponding metrics, in the data definition language (DDL) commands, which we will explain later. The syntax of the DML commands SELECT, UPDATE and DELETE are extended to allow expressing similarity predicates. However, here we describe only the constructions for the SELECT command, as the others are equivalent. The syntax of the INSERT command does not change, although its implementation does, as new complex objects need to be processed accordingly (as described in the 'Storing complex objects for similarity retrieval' section).

In order to illustrate the descriptions of the query commands, we consider that the following relation is already defined in the database:

```
CREATE TABLE Employee (
  Name CHAR(30),
  HomeLat FLOAT,
  HomeLongit FLOAT,
  Coordinate PARTICULATE,
  FrontalMugShot STILLIMAGE,
  ProfileMugShot STILLIMAGE,
  ... );
```

In addition to attributes of usual data types (e.g. Name, HomeLat and HomeLongit), this relation has three attributes of complex data types: Coordinate, FrontalMugShot and ProfileMugShot. As a PARTICULATE attribute, Coordinate is composed of attributes of numerical or categorical types already stored in the same relation, and in this example we assume that they are the HomeLat and HomeLongit ones. It is important to note that although in this example the particulate domain is being illustrated using a geographical data set, its utilization is not restricted to this data type, as is illustrated in the 'Example applications' section.

The SELECT command requires new constructions for similarity predicates in the WHERE and in the FROM clauses. The following subsections detail the SQL extensions that allow expressing all the similarity predicates described in the 'Fundamental concepts' section.

Similarity conditions in the WHERE clause

The simplest expression of a traditional, non-similarity-based predicate in the WHERE clause is a selection, which compares the set of values of an attribute with a constant value, in the format

DOI: 10.1002/spe

⟨*attr*⟩ θ ⟨*value*⟩, where θ is a relational or an exact match operator. A similarity-based selection follows the same format, where ⟨*attr*⟩ is a complex attribute attached to a metric, the constant ⟨*value*⟩ is an object in the same complex domain of the attribute and the operator θ is a similarity selection operator using a metric attached to the attribute. As similarity operators also require a similarity limit, we use the word NEAR to represent the operator, and another word to represent the limit; therefore, the proposed syntax to express a similarity-based selection is

<attr> NEAR <value> [STOP AFTER <k>] [RANGE <ξ>],

where STOP AFTER <k> indicates a k-nearest-neighbor selection and RANGE <ξ> indicates a range selection.

The ⟨*value*⟩ employed in a selection must be constant during the query execution. We allow three ways to express it: as an object in the computer's file system; directly as a constant in the query command; or as a reference to a value already stored in the database. As an illustration to select monolithic objects similar to an object stored in the file system, consider selecting the five employees with frontal mugshots most similar to the one given in the query. The following command answers it:

```
SELECT * FROM Employee
  WHERE FrontalMugShot NEAR 'c:\JohnFrontal.jpg'
  STOP AFTER 5;
```

If the attribute is PARTICULATE, a constant can be expressed as a list of [<value> AS <place_holder>], one for each parameter in the metric. For example, to select the employees living nearer than 2 units from a given coordinate expressed by a Latitude and a Longitude, the following command can be used:

```
SELECT * FROM Employee
 WHERE HomeCoordinate NEAR (-22.01 AS Latitude,
                            47.53 AS Longitude) RANGE 2;
```

As an example of the third way to express query values (retrieving them from the database), suppose we want to retrieve the 5 employees living nearer to the employee whose name is 'John Doe'. The following command retrieves them:

```
SELECT * FROM Employee
 WHERE HomeCoordinate NEAR (
   SELECT HomeLat AS Latitude, HomeLongit AS Longitude
     FROM Employee WHERE name='John Doe') STOP AFTER 5;
```

Note that, as HomeCoordinate PARTICULATE is composed of HomeLat and HomeLongit (as it will be explained later), the same query can be posed using the following command:

```
SELECT * FROM Employee
 WHERE HomeCoordinate NEAR (
   SELECT HomeCoordinate
     FROM Employee WHERE name='John Doe') STOP AFTER 5;
```

If name is not a key in the above command, then the inner, non-similarity-based SELECT can return more than one tuple. In this case the outer, similarity-based select receives a set of query

DOI: 10.1002/spe

centers as the query data set, characterizing an aggregate-similarity query, and an aggregate function must be chosen. As the function shown as Equation (1) in the 'Fundamental Concepts' section can handle a vast majority of needs, we restrict the proposed SQL extension to support just it, allowing to choose one of the three possible values for p, represented by a keyword following the operator NEAR. The choices are represented by the word SUM for $p=1$ (minimizes the sum of distances), ALL for $p=2$ (minimizes the average distance for all centers) or MAX for $p=\inf$ (minimizes the maximum distance). When no keyword is specified and more than one center is presented to the query, the default behavior is to assume SUM.

For example, to retrieve the names and frontal mugshots of the three employees whose home coordinate is nearer to the homes of the Doe's family, the following command can be issued:

```
SELECT Name, FrontalMugShot FROM Employee
 WHERE HomeCoordinate NEAR ALL (
   SELECT HomeCoordinate FROM Employee
     WHERE name like '% Doe') STOP AFTER 3;
```

A similarity join can be expressed in the WHERE clause too. Traditional joins are expressed comparing sets of values of attributes from two tables, in the format $R_1.attr_1\ \theta\ R_2.attr_2$. In the proposed extension, the similarity joins are expressed using the same format, provided both attributes are from the same complex domain and both are attached to a common metric. The construction $R_1.attr_1$ NEAR $R_2.attr_2$ RANGE ξ expresses a range join, the construction $R_1.attr_1$ NEAR $R_2.attr_2$ STOP AFTER k expresses a nearest join and the construction $R_1.attr_1$ NEAR ANY $R_2.attr_2$ STOP AFTER k expresses a closest join. For example, the following command retrieves, for each employee based in New York, the five most similar profiles of employees that are based in San Francisco:

```
SELECT * FROM Employee E1, Employee E2
 WHERE E1.FrontalMugShot NEAR E2.FrontalMugShot STOP AFTER 5
   AND E1.City = 'New York'
   AND E2.City = 'San Francisco';
```

This example shows that our approach also meets the second requirement to support similarity queries in SQL: '*How to enable the optimization of similarity queries*?' Although we do not discuss optimization issues in this paper, it is easy to see that the previous example allows using the optimization techniques used with traditional predicates: selection anticipation/conjunction with the join operation, employing existing index structures, and so on. Furthermore, more complex queries also provide more opportunities for optimization.

Similarity joins can also be expressed in the FROM clause, following the traditional join representation using a syntax equivalent to the one used in the WHERE clause. The FROM clause allows specifying all the parameters of the similarity joins, as shown in the 'Appendix'.

Variations on the basic commands can be expressed with modifiers in predicates involving both selections and joins, as follows. To retrieve the most dissimilar elements instead of the most similar elements, the word NEAR is replaced by the word FAR. Queries limited to k neighbors (in either selections or joins) can take into account the occurrence of ties. The default behavior of a k-limited query is retrieving just k elements, regardless of ties. Therefore, if two or more objects tie at the limiting distance, tying objects are randomly chosen to complete k objects in the answer. However, this behavior can lead to non-repeatable answers. An alternative is to specify WITH TIE LIST following the STOP AFTER keyword. In this case all tying objects are returned, possibly increasing

DOI: 10.1002/spe

the number of objects returned. The following example asks for the 10 objects more distant from the query center but allows retrieving more if ties occur at the 10th position:

```
SELECT * FROM Employee
  WHERE FrontalMugShot NEAR 'c:\JohnFrontal.jpg'
  STOP AFTER 10 WITH TIE LIST;
```

Both `STOP AFTER` and `RANGE` can be specified in the same query. In this case, an implicit conjunction is assumed, limiting the answer to have at most k objects not farther (or not nearer) than the distance ξ from the query center. For example, the command

```
SELECT * FROM Employee
  WHERE FrontalMugShot NEAR 'c:\img.jpg'
    STOP AFTER 5 RANGE 0.03;
```

retrieves at most five images not farther than 0.03 units from the image given.

It is important to note that the presented constructions are powerful enough to cover the full set of similarity operators presented in the 'Fundamental Concepts' section, yet they require only the addition of two more predicates (`near` and `far`), which are integrated with the already existing ones. Therefore, the proposed syntax allows seamlessly including similarity queries in SQL, meeting the first requirement stated in the 'Introduction'.

The `CREATE METRIC` command

The third and last requirement to support similarity queries in SQL asks '*How to support any number of user-defined feature extractors and the corresponding metrics over complex data so that we can compare objects?*' In fact, there is no concept related to the definition of comparison operators in SQL; hence, it is a new resource that must be included, and new commands are required to support it. Our conceptual basis to support similarity queries is employing distance functions—or metrics—as the mechanism to compare pairs of complex objects. Therefore, we define three new commands to handle metrics: the `CREATE METRIC`, the `ALTER METRIC` and the `DROP METRIC` commands. According to the complex domain the `ALTER METRIC` command performs a specific task. Considering monolithic domains it allows the inclusion or removal of features that should be used to compare pairs of complex objects and considering particulate domains it allows one to add or remove the attributes that compose the distance function. The `DROP METRIC` command can be used to erase a specified metric. Here we describe only the `CREATE METRIC` command, as the others follow similar constructions. Note that those are the only new commands needed to support similarity queries, as the other presented modifications are just extensions on existing commands.

The definition of a metric must be stored in the database catalog; thus, its corresponding manipulation commands follow the DDL command style. As a metric is defined over a complex domain, it can be attached to any attribute of the data types from the corresponding domain. Therefore, before comparing elements of a given attribute following a given way, two requirements must be met:

1. create a metric;
2. attach the metric to the attribute.

The distinction we made among particulate and monolithic domains affects the way metrics are defined and thus there are two variations of the `CREATE METRIC` command: one for

DOI: 10.1002/spe

PARTICULATE and another for MONOLITHIC domains. A PARTICULATE attribute is defined over other attributes (numbers or small texts) stored in the same tuple in the relation; therefore, two values of a given PARTICULATE domain are a function of their constituting attributes. A MONOLITHIC attribute is stored as a large binary object, and it can only be compared extracting a set of representative features that can identify each object, which is used in place of the real object when it needs to be compared. Therefore, the specification of a metric for MONOLITHIC domains must include the definition of the feature extractors employed and the definition of how the features compose the distance function, whereas the specification of a metric for PARTICULATE domains just includes the definition of how the constituting attributes compose the distance function. Note that in both cases the definition of a metric is generic, as it will be later attached to any attribute of the corresponding domain.

The syntax of a CREATE METRIC command for PARTICULATE domains is

```
CREATE METRIC <name> USING <function> FOR PARTICULATE
  (<place_holder> <data_type>[, <place_holder> <data_type>]...);
```

where the place_holders represent the parameters of the distance function with its respective data_types, and function is the basic function that must be employed. In the proposed extension, we support the Minkowski distance of order one, two or infinity; hence, function can be either L1, L2 or L∞, for the City Block, Euclidean or Chebyshev distance.

For example, consider the following command:

```
CREATE METRIC GeoDistance USING LP2 FOR PARTICULATE
  (Latitude FLOAT, Longitude FLOAT);
```

This command specifies a metric called GeoDistance, which says that two PARTICULATE values have their dissimilarity evaluated using the Euclidean distance (L2) based on two parameters of type FLOAT: Latitude and Longitude.

The comparison of elements from MONOLITHIC domains requires to extract their features before the metric can be applied. Therefore, the corresponding CREATE METRIC commands must specify the feature extractors that must be executed in order to obtain the features needed. Each feature extractor is specific for its domain. For example, an image feature extractor applies only over STILLIMAGE objects, and it is typically an image processing algorithm. Each application must define its own feature extractors, as procedures that receive a complex object as an argument, and return any number of features meaningful to compare complex data.

The syntax of a CREATE METRIC command for MONOLITHIC domains is

```
CREATE METRIC <name> USING <function> FOR STILLIMAGE | AUDIO | ...
  (<extractor>(<feature> AS <place holder>,...),
   <extractor>(<feature> AS <place holder>,...) ...);
```

where function is similar to its counterpart for PARTICULATE domains and each extractor represents a feature extractor. Each feature extractor can extract several features, but only those indicated between the corresponding parenthesis will be used by this metric. Each useful feature assumes the place indicated by the corresponding place_holder in the distance function.

Suppose an extractor called HistogramEXT() was defined returning an array of integers representing the color histogram of an image. The following command shows how to create a metric

DOI: 10.1002/spe

called Histogram to compare images regarding color histograms:

```
CREATE METRIC Histogram USING LP1 FOR STILLIMAGE
  (HistogramEXT (HistogramFeature AS Histo));
```

Attaching metrics to complex data types

A metric must be attached to attributes in order to allow comparisons. A metric can be attached to several attributes in any relation, provided the attributes and the metric are of the same complex data type. Metrics are attached to attributes as an attribute constraint or as a table constraint, following the usual ways to define constraints in a CREATE TABLE command. Moreover, as metrics enable the creation of indexes to speedup queries, they can also be defined in a CREATE INDEX command.

Attaching a metric to a particulate attribute requires to associate each of its constituting attribute with the corresponding place_holder of the metric. The syntax to specify a table constraint to a PARTICULATE attribute in a CREATE METRIC command is

```
METRIC (<Particulate_name>)
       REFERENCES (<attr1> AS <param1>, <attr2> AS <param2>, ...)
       USING (metric_name)
```

The REFERENCES clause associates each attribute $attr_i$ composing the particulate object Particulate_name with each corresponding place_holder $param_i$ used in the metric metric_name. Note that, as a PARTICULATE attribute is composed of other attributes already stored in the relation, it does not present a 'value' by itself. Rather, the PARTICULATE attribute is a reference to the set of values stored in its constituting attributes.

For example, the following table constraint allows comparing HomeCoordinates in the Employee relation using the GeoDistance metric.

```
 ...
METRIC (HomeCoordinate)
  REFERENCES (HomeLat AS Latitude, HomeLongit AS Longitude)
  USING (GeoDistance),
 ...
```

An equivalent column constraint can be used, provided the composing attributes are defined prior to the particulate attribute, as shown in the 'Appendix'.

A monolithic object is stored as a single value, and the parameters used in its comparisons are retrieved by feature extractors specified in each metric. Therefore, attaching a metric to a MONOLITHIC attribute only requires declaring that the metric can be used with the attribute. Monolithic domains do not need the REFERENCES clause because the associations are implicit in the feature extractor declarations in the metric. Several metrics can be associated with the same complex attribute, and in this case the first attachment is the default one, unless the word DEFAULT is specified.

As an example, let us consider the Employee relation, where both FrontalMugShot and ProfileMugShot attributes are images. Suppose that the metrics Histogram and Texture were previously defined over STILLIMAGE domains. If we want to attach the Histogram metric to both FrontalMugShot and ProfileMugShot attributes and the Texture metric

DOI: 10.1002/spe

to ProfileMugShot, the following column constraints can be declared:

```
CREATE TABLE Employee (
             Name CHAR(30),
             FrontalMugShot STILLIMAGE METRIC USING (Histogram),
             ProfileMugShot STILLIMAGE METRIC USING (Histogram, Texture),
             ... );
```

Note that, as both FrontalMugShot and ProfileMugShot attributes share Histogram as a common metric, thus in addition to comparing the set of values of the FrontalMugShot and the ProfileMugShot attributes among themselves, at least syntactically, a set of values of the FrontalMugShot attribute can also be compared with a set of values of the ProfileMugShot attribute. However, only sets of values of ProfileMugShot attributes can be compared using the Texture metric.

To employ a metric other than the default one, the clause BY <metric name> is used in the query predicates. For example, the following command selects up to five employees whose profile mugshots are the most similar to a given one considering both Histogram and Texture metrics:

```
SELECT * FROM Employee
  WHERE (ProfileMugShot NEAR 'c:\img.jpg' BY Histogram STOP AFTER 5) AND
        (ProfileMugShot NEAR 'c:\img.jpg' BY Texture STOP AFTER 5);
```

Creating indexes on complex data types

Besides constraints in a CREATE TABLE command, another option to attach metrics to attributes is using the CREATE INDEX command. This option not only declares the attachment but also explicitly asks the DBMS to use a MAM to create an index for it. The following are examples of creating an index for the HomeCoordinate attribute and an index for the FrontalMugShot attribute in the Employee relation.

```
CREATE INDEX HomePosition ON Employee(HomeCoordinate)
  REFERENCES (HomeLat AS Latitude, HomeLongit AS Longitude)
  USING GeoDistance;
CREATE INDEX Frontal ON Employee (FrontalMugShot)
  USING Histogram;
```

Storing complex objects for similarity retrieval

To enable posing similarity queries on complex objects, the data dictionary of the DBMS must be extended with parameters that describe these objects. They are employed by SIREN to manipulate information such as

- which feature extractors are employed by each metric defined for the STILLIMAGE and AUDIO data types;
- which traditional attributes compose each PARTICULATE attribute;
- which metrics are defined for each complex domain;

DOI: 10.1002/spe

- which are the complex attributes stored in each relation; and
- which metrics are associated with each complex attribute.

The parameters are obtained when SIREN interprets the DDL commands to create indexes and to create indexes and tables that make references to or that contain attributes of complex types.

Besides the meta-data stored in the data dictionary, each MONOLITHIC object also requires storing specific information. Particulate data types are stored by their constituent parts; hence, they present no new storage requirements besides the ones already available in the DBMS. On the other hand, MONOLITHIC objects require storing the features that are extracted from them, besides the Binary Large Objects (the BLOB value). This is required because the feature extraction processes are usually costly; hence, they should be executed only once for each object, when the object is stored in the database.

The features are usually stored either as categorical or as numeric attributes and are associated with the complex object. As the user does not provide space in his/her relation to store the extracted features, the system must provide both the required space and ways to associate them with the BLOB data in a way completely transparent to the user. This is performed by SIREN intercepting the CREATE TABLES for the user tables that contain STILLIMAGE/AUDIO attributes, changing their structure as follows. Each complex attribute is replaced by a reference to a system table, which has as attributes a BLOB attribute to store the object, an internal identifier for the reference and attributes to store the features extracted by every extractor used in all metrics attached to the attribute. There is one system table for each STILLIMAGE/AUDIO attribute in each user table. Thereafter, the user sees a view joining the system tables and the changed user table, hiding the identifier and the feature attributes; hence, the user sees the table as it was defined.

Whenever a new tuple containing an image/audio track is stored in the database, SIREN intercepts the INSERT command, stores the set of values of non-image attributes in the user table and the image/audio track as a BLOB in the corresponding system table. Then, for each STILLIMAGE/AUDIO attribute, SIREN calls all feature extractors attached to the attribute and stores the extracted features in the corresponding system table. When the user poses queries involving similarity predicates, SIREN uses the extracted features to execute the similarity operations, and then it joins the results with those obtained from the non-similarity-based part of the query using the underlying DBMS.

Figure 4 shows an example of how the new complex data types are stored by SIREN. The MONOLITHIC types STILLIMAGE and AUDIO require storing the BLOB values in the system tables with the corresponding extracted features. Note that the complex attributes in the user table are replaced by references to the corresponding system tables. The PARTICULATE attributes do not require system tables. The value of a PARTICULATE attribute, as for every complex attribute, is a reference to the tuple itself.

Similarity query processing

Here we illustrate how SIREN executes an SQL similarity command, using the following SELECT command. It contains a single similarity predicate, centered at an object stored outside the database.

```
SELECT Name, FrontalMugShot FROM Employee
  WHERE FrontalMugShot NEAR 'c:\JohnFrontal.jpg'
  STOP AFTER 5;
```


DOI: 10.1002/spe

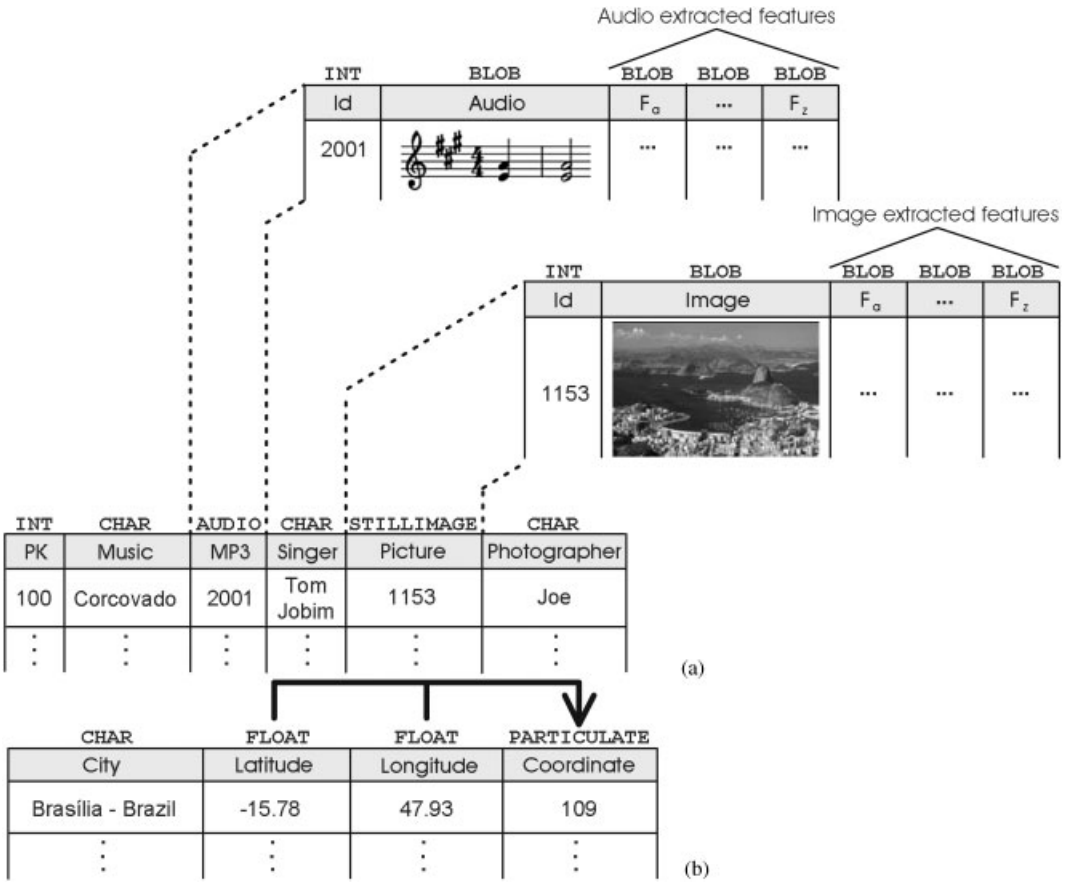

Figure 4. Data structure to store complex data: (a) attributes of `MONOLITHIC` data types and (b) attributes of the `PARTICULATE` data type.

This command is analyzed and rewritten by SIREN following the steps showed in Figure 5, as described below.

1. Initially, the application program submits the SQL command.
2. The command interpreter analyzes the original command, identifying the type of similarity operation (`NEAR`) that needs to be executed (*kNN*), the complex attribute involved (`FrontalMugShot`) and the parameters of the predicate (single query center s_q = `'c:\JohnFrontal.jpg'`, $k = 5$ and metric = attribute's default). Thereafter, it queries the SIREN data dictionary.
3. The SIREN data dictionary is searched to obtain information regarding the complex attribute `FrontalMugShot`: which is the attribute's default metric (default = `Histogram`), which feature extractors are employed by the metric (the histogram extractor), the distance function to be used (L_1), and the index structure *Ix* to be employed.

DOI: 10.1002/spe

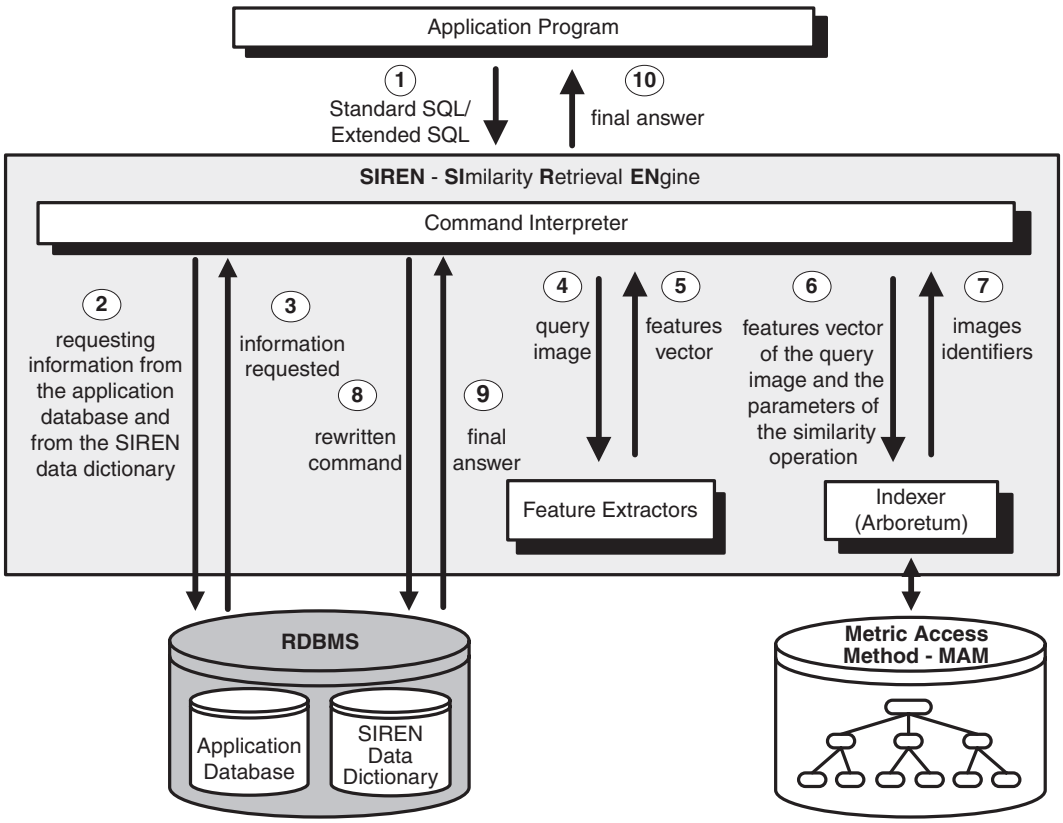

Figure 5. An example of the command execution process.

4. The query image s_q is submitted to the required feature extractor (in this example just the histogram extractor).
5. The extracted feature vector V is returned.
6. The interpreter sends to the indexer the following parameters: the feature vector V, the similarity operation (*kNN*) and its respective parameters (s_q, k), and the index structure Ix.
7. The indexer returns the set of image identifiers S_{Oid} that answers the posed command.
8. The command interpreter uses the identifiers S_{Oid} to rewrite the original command submitted by the application program. The command rewritten by SIREN for the current example is presented as follows:

```
SELECT Name, IPV$Employee_FrontalMugShot.Image AS FrontalMugShot
  FROM Employee JOIN IPV$Employee_FrontalMugShot
       ON Employee.FrontalMugShot = IPV$Employee_FrontalMugShot.Image_id
 WHERE FrontalMugShot IN (7896, 7912, 9669, 9668, 9675);
```


DOI: 10.1002/spe

where (7896, 7912, 9669, 9668, 9675) are examples of identifiers returned. The rewritten command is submitted to the DBMS.
9. The DBMS answers the query, obtaining the images and the traditional attributes requested.
10. The answer returned by the DBMS is sent to the application program.

It is worth noting that if a query image stored in the DBMS was specified in the original command submitted by the application program, it would not be sent to the feature extractors. In this case, Steps 4 and 5 are replaced by a query to the DBMS in order to obtain the feature vector of the stored image.

ANSWERING SIMILARITY QUERIES IN SIREN

The main issue of defining a new extractor in SIREN is the development of the image/audio processing algorithms that extract the desired features. In fact, integrating an existing feature extractor to SIREN is simple. The literature on image and audio processing has several feature extractors already described. The current version of SIREN includes three image feature extractors: a texture extractor (TEXTUREEXT) [18], a shape extractor based on Zernike moments (ZERNIKEEXT) [19] and a color extractor based on the normalized color histogram (HISTOGRAMEXT) [20]. There is also one extractor for the AUDIO data type: a sound-texture extractor (SOUNDTEXTUREEXT) that extracts the Mel-frequency cepstral coefficients (MFCC) and features based on the short time Fourier transform (STFT) [21,22]. SIREN has been implemented in C++ and it employs the ODBC protocol to connect to a DBMS. Currently the prototype runs over Oracle 10g or PostgreSQL 8.2. Both share an identical query language extension.

SIREN implements algorithms to execute all the main similarity operators: two similarity selections for single query centers, two selections for multiple centers and three similarity joins. The traditional single center selections, that is, the similarity range query and the k-nearest-neighbor query operators, are available in several MAMs. SIREN employs the Slim-tree MAM [8] to index the complex attributes. The implementation of Slim-tree was taken from Arboretum, an open-source C++ library that implements various MAMs [23]. Unfortunately, there is no procedure yet to execute the other five operators in any published MAM. Therefore, we implemented the remaining five procedures using the sequential scan method to validate both SIREN and our approach to support similarity queries in SQL. Thus, when a complex attribute is associated with an index, the single center similarity selections are executed using the Slim-tree. The other operations and all operations over complex attributes that do not have an index are answered using sequential scan.

The time required to answer a query varies depending on the query operator requested and on its parameters. The issues involved in the execution of each similarity operator are not the concern of this paper. However, to provide an intuition of the execution times involved, as well as to show that the state of the art of similarity retrieval techniques is ready to support similarity queries in SQL, we measured the times required in processing some of the implemented procedures. Note that searching both monolithic and particulate data types relies upon values stored as numerical attributes in one table; hence, there is no procedural difference in performing similarity search on both data types.

DOI: 10.1002/spe

The measurements were performed on a data set R_1 of 40 000 medical images from computerized tomographies (CT) from several human body parts. The metric is the Manhattan (L_1) distance function over normalized gray-scale histograms. For the similarity join test, another table was created with 400 images randomly selected from the data set R_1 to create data set R_2.

Considering the procedures implemented in the Slim-tree MAM, the execution of the range (with $\xi = 0.2$) and k-nearest-neighbor (with $k = 10$) selection operations required 0.26 and 0.36 s, respectively. As a reference of one of the most expensive operations among the ones performed using sequential scan, the execution of the k-nearest-neighbors join ($R_2 \overset{kNN}{\bowtie} R_1$ with $k = 10$) required 241 s.

The tests were performed on a Windows XP platform, using a Pentium D CPU of 3.4 GHz, 2 GB of main memory and a 250 GB Serial ATA. The database was managed by the Oracle 10g Express Edition. The total time required by the procedures tested corresponds to the average time in seconds spent to compile the command and to execute the similarity operators. They do not include the time to extract the image features, access the images themselves or their rendering in the user interface. It is important to note that the time spent by SIREN to compile the command did not exceed 0.10 s for all the procedures tested. As it can be seen, although the most demanding queries could not rely on an index structure, the results are obtained in acceptable time, bringing similarity queries as a viable tool to gather information from a database.

EXAMPLE APPLICATIONS

In order to show the usability of the SIREN, a web (cgi) application was developed in C++. This application provides an environment for users to pose direct SQL commands. It is employed here to explore two data sets: the first data set is called `MedImages`, and is composed of 5180 medical images of CT from three human body parts: abdomen, cranium and thorax. Each tuple of this data set is constituted by an image id, the image, the body part identifier and an attribute stating whether or not the image identifies a pathological condition.

Similarity queries over the `MedImages` data set can explore several aspects of the images, such as similarity based on color distribution or on texture. Two metrics were defined using two of the three implemented extractors over images to handle these features: one using the `HISTOGRAMEXT` extractor and the other using the `TEXTUREEXT` extractor. The commands employed to create the table and image metrics are as follows:

```
CREATE METRIC MyHistogram USING LP1 FOR STILLIMAGE
       (HISTOGRAMEXT (Histogram AS Histo));
CREATE METRIC MyTexture FOR STILLIMAGE
       (TEXTUREEXT (Texture AS T));
CREATE TABLE MedImages (
       Id INTEGER PRIMARY KEY,
       BodyPart VARCHAR(15),
       Img STILLIMAGE,
       Pathology CHAR(1),
       METRIC (Img) USING (MyHistogram DEFAULT, MyTexture));
```


DOI: 10.1002/spe

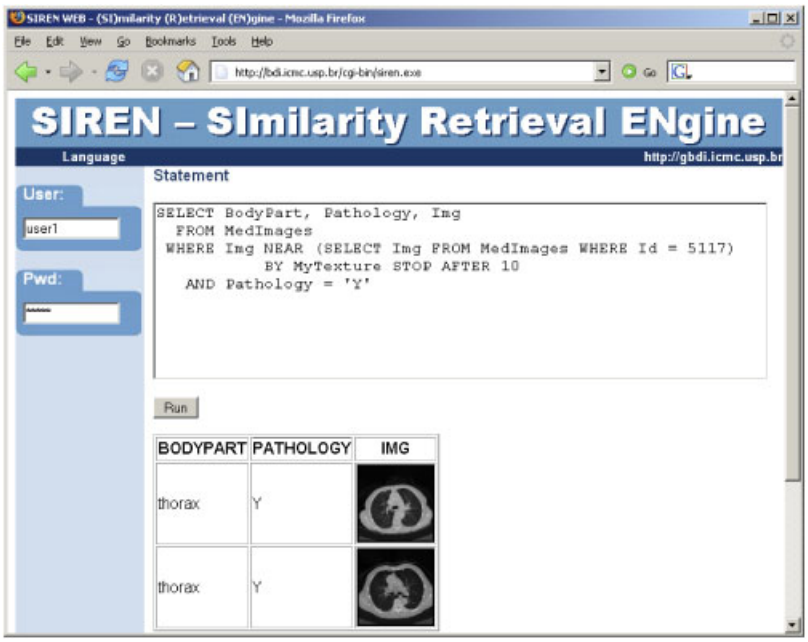

Figure 6. Similarity query examples based on `MedImages` table. *k*-NN query considering the `MyTexture` metric.

The command shown in Figure 6 was submitted to SIREN to select up to 10 images presenting a pathology that are similar to the given image of the database, regarding their texture similarity. In this example, the query image was obtained through a subquery (nested) command, the metric employed was the `MyTexture` and a non-similarity predicate was also used (`Pathology='Y'`). Consider now the question 'Which are the most similar images not farther than 0.2 units from a given image, considering the similarity of their color histograms?'. The corresponding command is shown in Figure 7, as submitted to SIREN. The query image was specified as a path in the file system, and as the metric required was the default one (`MyHistogram`), its specification was omitted.

The second data set is employed to illustrate the `PARTICULATE` data type. It is called `Cars`[§] and is composed of 392 car descriptions. This data set has five attributes: MPG (miles per gallon), horsepower, time to accelerate from 0 to 60 mph (s), car origin (American, European or Japanese) and the car name. The data set could be explored by similarity queries considering several metrics. Here we show a metric to compare cars based on what we call the cost-benefit ratio related to the variables horsepower, acceleration and MPG. The commands employed to create the table and the

[§] This is the cars.data data set available at http://lib.stat.cmu.edu/.

DOI: 10.1002/spe

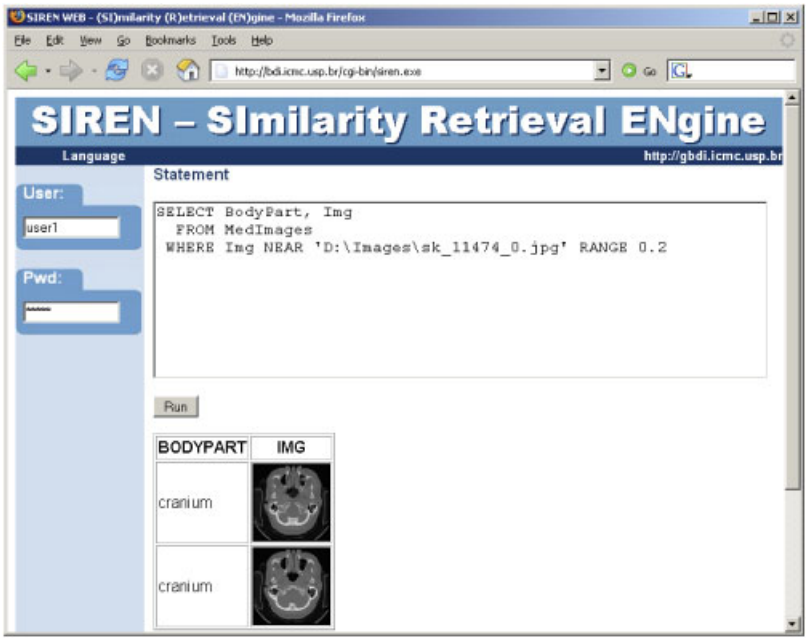

Figure 7. Similarity query examples based on `MedImages` table. Range query considering the default metric (`MyHistogram`).

metric are the following:

```
CREATE METRIC CostBenefit USING LP2 FOR PARTICULATE
       (hp FLOAT 5.0, mpg FLOAT, sec FLOAT 10.0);

CREATE TABLE Cars (
       CarName CHAR(35),
       Horsepower FLOAT,
       Consumption FLOAT,
       Acceleration FLOAT,
       Origin CHAR(8),
       Car PARTICULATE,
       METRIC REFERENCES (Horsepower AS hp,
                          Consumption AS mpg,
                          Acceleration AS sec)
              USING (CostBenefit DEFAULT));
```

Using the `CostBenefit` metric, it is possible to perform queries such as: 'Which are the 4 most similar cars having: horsepower = 95 hp, consumption = 24 mpg and acceleration = 15 s?' and

DOI: 10.1002/spe

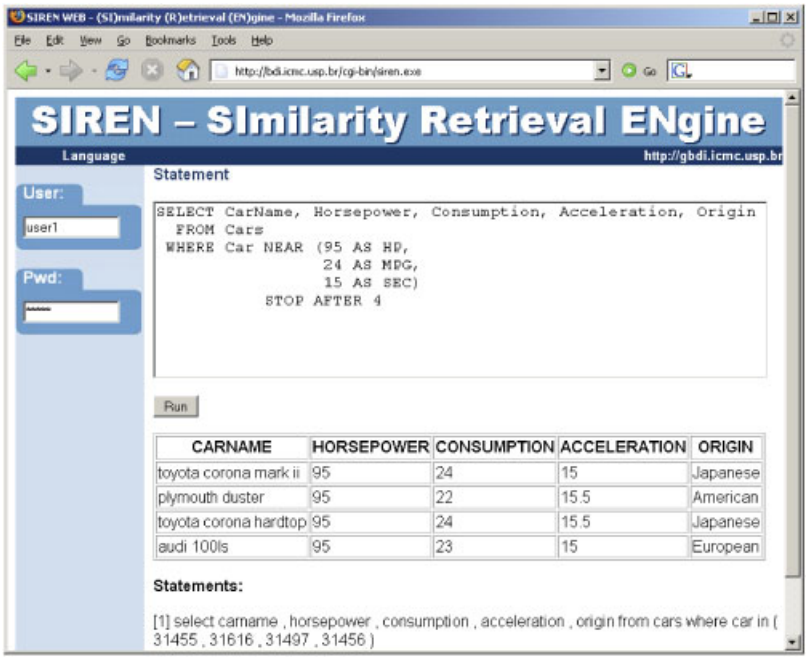

Figure 8. Similarity query examples over the `Cars` table: 'Which are the 4 most similar cars having: horsepower = 95 hp, consumption = 24 mpg and acceleration = 15 s?'.

'Which are the 2 European cars most similar to each American car?' These two queries are presented in Figures 8 and 9, respectively.

CONCLUSIONS

The problems occurring when searching large sets of complex objects by similarity have attracted the attention of many researchers in the database and machine-learning communities. However, each work developed until now focuses on a specific topic related to the broad concept of similarity, and there is no conducting line able to put together these many disperse efforts. Supporting similarity queries in SQL, enabling the seamless integration of similarity queries with the other resources of the language, can be that conducting line.

The support for similarity queries presented herein is powerful enough to allow several predicate variations, including single and multiple centers similarity selections and similarity joins. The predicates can be applied over any complex object for which a similarity measure can be defined, including large monolithic objects stored as blobs, as well as over objects stored as a particulate set of composing attributes. The presented approach also enables associating feature extractors with metrics over complex objects, enabling content-based retrieval of complex objects. Moreover, as complex objects often can be compared through several distinct perspectives, our approach allows us to attach any number of metrics to a complex attribute. We also report results from using a prototype

DOI: 10.1002/spe

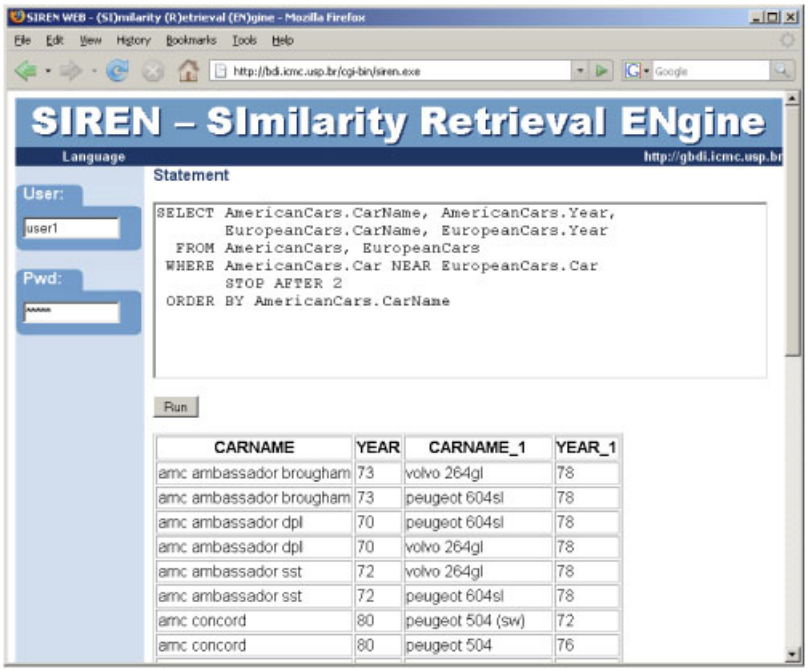

Figure 9. Similarity query examples based on the `Cars` table: 'Which are the 2 European cars most similar to each American car?'.

that implements similarity queries on images and particulate data using Oracle or PostgreSQL as the database server.

As a conducting line to put together research activities, our solution for similarity query representation also has several interesting characteristics. First, it enables representing similarity queries as just one more type of predicate, leading to the integration of similarity as operations in relational algebra. This characteristic will enable extending the optimizers of the relational DBMS to treat and optimize similarity queries as well. Second, the presented solution can benefit from improvements on data retrieval techniques aimed at similarity, such as the development of index structures that support similarity operators. This characteristic can also guide the development of such structures, as it directs to the types of retrieval operations that are worth improving, such as the aggregate similarity and similarity join operations. Third, the presented solution can act as a hub for the development of algorithms to perform broadly employed similarity operations regarding data analysis. For example, data mining processes often require performing similarity operations, and having them integrated in the database server, possibly optimized by a MAM, can be feasible in the future.

Similarity of complex objects is a broad research area involving the development of such diverse activities as the development of techniques to process each object (images, sound, etc.), to extract its features, to compare some of them, to index them, etc. Many efforts are currently being taken in the development of these techniques, but currently there is not yet an easy way to integrate those efforts. The language constructions presented in this paper can provide a proper integration,

DOI: 10.1002/spe

helping to boost even more these challenging and long-needed support for similarity retrieval in large collections of complex objects.

APPENDIX A: SYNTAX FOR SIMILARITY QUERIES IN SQL

This appendix presents the detailed syntax for similarity query support in SQL. The use of [ ] means a choice of optional terms, and { } means a choice of required terms. The complex data type can be either PARTICULATE or one of the MONOLITHIC ones. In this appendix we consider only AUDIO and STILLIMAGE as the monolithic ones, as they were already implemented in SIREN. However, other complex monolithic data types should share the same syntax. Regarding the DDL commands, we present here only the syntax of the CREATE METRIC, as the ALTER and DROP commands follow the same structure.

The CREATE METRIC command

The syntax to define a metric (that is, a similarity measure) is

```
CREATE METRIC <metric_name>
  [USING {LP0 | LP1 | LP2}] FOR
     {PARTICULATE '('<parameter_pref_list>')'
     |STILLIMAGE '('<extractor_ref_list>')'
     |AUDIO '('<extractor_ref_list>')'}
<extractor_ref_list>::= <extractor_ref>
    |<extractor_ref>',' <extractor_ref_list>
<extractor_ref>::= <extractor_name>
    '('<parameter_mref_list>')'
<parameter_mref_list>::= <parameter_mref>
   |<parameter_mref>','<parameter_mref_list>
<parameter_mref>::= <param_name> AS <param_alias> [<weight>]
<parameter_pref_list>::= <parameter_pref>
   |<parameter_pref>','<parameter_pref_list>
<parameter_pref>::= <param_name> <param_type> [<weight>]
```

Specifying METRIC as a column constraint

The METRIC constraint can be associated with STILLIMAGE, AUDIO or PARTICULATE data types. The data type of the attribute and the metric must be the same. The syntax to specify it as a column constraint is

```
<column_constraint>::= [<constraint_name>]
      NULL | PRIMARY KEY | ...
      | METRIC [ REFERENCES '('<param_assoc_list>')']
        USING '('<metric_name_list>')'
<param_assoc_list>::= <param_assoc>
```


DOI: 10.1002/spe

```
      | <param_assoc>',' <param_assoc_list>
<param_assoc>::= <attr_name> AS <param_name>
<metric_name_list>::= <metric_name> [DEFAULT]
      | <metric_name> [DEFAULT] ','
        <metric_name_list>
<metric_name>::= <stillimage_metric_name>
      | <particulate_metric_name>
```

The optional clause REFERENCES '('⟨param_assoc_list⟩')' is used to define a column constraint only for PARTICULATE attributes.

To be referenced in a similarity predicate, attributes of a complex data type must be associated with at least one metric. One metric can be associated with any number of complex attributes. If a complex attribute is associated with two or more metrics, then a default metric should be specified.

Specifying METRIC as a table constraint

The syntax to associate a metric with a complex attribute as a table constraint is

```
<table_constraint>::= [<constraint_name>]
     PRIMARY KEY | ...
     | METRIC {
       '('<stillimage_attr_name_list>')'
       | '('<audio_attr_name_list>')'
       | '('<particulate_attr_name_list>')'
         REFERENCES '('<param_assoc_list>')'
       } USING '('<metric_name_list>')'
```

CREATE INDEX for metric indexes

The syntactical constructions employed in the CREATE INDEX command for the specification of indexes over complex data attributes are presented as follows:

```
CREATE INDEX <index_name>
     ON <table_name> {
        '('<stillimage_attr_name>')'
        | '('<audio_attr_name>')'
        | '('<particulate_attr_name>')'
          REFERENCES (<param_assoc_list>)
     } USING <metric_name> [DEFAULT]
```

Specifying similarity queries in the SELECT command

The SELECT command is extended with a new construction to specify any similarity predicates in the WHERE clause, and another to specify similarity joins in the FROM clause.

DOI: 10.1002/spe

Similarity predicates in the `WHERE` *clause*

Similarity selection queries are always expressed as predicates in the `WHERE` clause. Similarity joins are expressed as a predicate either in the `WHERE` clause or in the `FROM` clause.

```
<similarity_predicate>::=<complex_attr_name1>
       {NEAR | FAR}
       [<similarity_aggregation>]
       { <complex_attr_name2> | <attr_value> |
         '('<attr_value_set>')' }
       [BY <metric_name>]
       [STOP AFTER <k> [WITH TIE LIST] ]
       [RANGE <ξ>]
<similarity_aggregation>::= SUM | ALL | MAX
```

The set of values of the attribute <complex_attr_name$_1$> is the one to be searched in the comparison predicates. It can be compared with a constant value <attr_value>, with a set of constant values '('<attr_value_set>')' or with a set of values of another attribute in any table of the database. If attribute <complex_attr_name$_1$> is `STILLIMAGE` or `AUDIO` and must be compared with a constant value, the constant is an image/audio track expressed as a path in the file system where the image/audio track is stored. If attribute <complex_attr_name$_1$> is `PARTICULATE`, the constant is expressed as

```
<attr_value>::='('<param_val_assoc_list>')'
<param_val_assoc_list>::= <param_val_assoc>
 |<param_val_assoc>',' <param_val_assoc_list>
<param_val_assoc>::= <attribute> AS <param_name>
```

Comparing the set of values of the <complex_attr_name$_1$> with a constant value or with a set of constants represents a similarity selection. Comparing the set of values of the attribute with the set of values of another attribute in the same domain represents a similarity join. Specifying both `STOP AFTER` and `RANGE` clauses requires meeting both k and ξ limits.

If neither `STOP AFTER` nor `RANGE` is specified, then `RANGE 0` (zero) is assumed. The construction <complex_attr_name$_1$> `NEAR|FAR` <complex_attr_name$_2$> `STOP AFTER` k represents a nearest join. The construction <complex_attr_name$_1$> `NEAR|FAR ANY` <complex_attr_name$_2$> `STOP AFTER` k represents a closest join. The construction <complex_attr_name$_1$> `NEAR|FAR` <complex_attr_name$_2$> `RANGE` ξ represents a similarity range join. The construction <complex_attr_name$_1$> `NEAR|FAR` <attr_value> represents either *kNN* or similarity range select queries. The construction <complex_attr_name$_1$> `NEAR|FAR '('`<attr_value_set>`')'` represents an aggregate select query. If the <similarity_aggregation> clause is omitted in a similarity query having more than one query center, `SUM` is assumed. If the `BY` clause is omitted, the default metric is assumed. If `WITH TIE LIST` is omitted in a `STOP AFTER` clause, no tie list is assumed.

DOI: 10.1002/spe

Similarity joins in the `FROM` *clause*

The syntax to express similarity joins in the `FROM` clause is

```
<joined_table>::=
    <table1> <sim_join_type> <table2>
        ON <complex_attr_name1> {NEAR | FAR}
           <complex_attr_name2>
        [STOP AFTER <k>]
        [RANGE <ξ>]
<sim_join_type>::= {CLOSEST | NEAREST | RANGE} JOIN
```

If `CLOSEST JOIN` or `NEAREST JOIN` is specified but not `STOP AFTER`, then k is assumed to be 1. If `RANGE JOIN` is specified but not `RANGE`, then ξ is assumed to be 0, returning those objects occurring in both tables.

REFERENCES

1. Ciaccia P, Patella M. Searching in metric spaces with user-defined and approximate distances. *ACM Transactions on Database Systems* 2002; **27**(4):398–437. DOI: 10.1145/582410.582412.
2. Roussopoulos N, Kelley S, Vincent F. Nearest neighbor queries. *Proceedings of the ACM International Conference on Management of Data* (*SIGMOD*), San Jose, CA, 1995; 71–79. DOI: 10.1145/223784.223794.
3. Hjaltason GR, Samet H. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems* 2003; **28**(4):517–580. DOI: 10.1145/958942.958948.
4. Böhm C, Krebs F. The k-nearest neighbour join: Turbo charging the KDD process. *Knowledge and Information Systems* 2004; **6**(6):728–749. DOI: 10.1007/s10115-003-0122-9.
5. Barioni MCN, Razente H, Traina AJM, Traina C Jr. SIREN: A similarity retrieval engine for complex data. *Proceedings of the 32nd International Conference on Very Large Data Bases* (*VLDB*), Seoul, South Korea, 2006; 1155–1158.
6. Barioni MCN, Razente H, Traina C Jr, Traina AJM. Querying complex objects by similarity in SQL. *Proceedings of the 20th Brazilian Symposium on Databases* (*SBBD*), Uberlandia, Brazil, 2005; 130–144.
7. Ciaccia P, Patella M, Zezula P. M-tree: An efficient access method for similarity search in metric spaces. *Proceedings of the 23rd International Conference on Very Large Data Bases* (*VLDB*), Athens, Greece, 1997; 426–435.
8. Traina C Jr, Traina AJM, Faloutsos C, Seeger B. Fast indexing and visualization of metric datasets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering* 2002; **14**(2):244–260. DOI: 10.1109/69.991715.
9. Carey MJ, Kossmann D. Reducing the braking distance of an SQL query engine. *Proceedings of the 24th International Conference on Very Large Data Bases* (*VLDB*), New York City, NY, 1998; 158–169.
10. Gao L, Wang M, Xiaoyang SW, Padmanabhan S. Expressing and optimizing similarity-based queries in SQL. *Proceedings of the 23rd International Conference on Conceptual Modeling* (*ER*) (*Lecture Notes in Computer Science*). Springer: Berlin, 2004; 464–478.
11. Melton J, Eisenberg A. SQL multimedia and application packages (SQL/MM). *SIGMOD Record* 2001; **30**:97–102. DOI: 10.1145/604264.604280.
12. Traina C Jr, Traina AJM, Vieira MR, Arantes AS, Faloutsos C. Efficient processing of complex similarity queries in RDBMS through query rewriting. *Proceedings of the 15th ACM Conference on Information and Knowledge Management* (*CIKM*). ACM: New York, 2006; 1–10.
13. Ferreira MRP, Traina C Jr, Traina AJM. An efficient framework for similarity query optimization. *Proceedings of the 15th ACM International Symposium on Advances in Geographic Information Systems* (*ACM GIS*), Seattle, WA, 2007; 396–399.
14. Smeulders AWM, Worring M, Santini S, Gupta A, Jain R. Content-based image retrieval at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 2000; **22**(12):1349–1380. DOI: 10.1109/34.895972.
15. Papadias D, Shen Q, Tao Y, Mouratidis K. Group nearest neighbor queries. *Proceedings of the 20th IEEE Computer Society International Conference on Data Engineering* (*ICDE*), Boston, MA, 2004; 301–312. DOI: 10.1109/ICDE.2004.1320006.

16. Doulamis N, Doulamis A. Evaluation of relevance feedback schemes in content-based in retrieval systems. *Signal Processing*: *Image Communication* 2006; **21**(4):334–357. DOI: 10.1016/j.image.2005.11.006.
17. GBDI-ICMC-USP. Similarity retrieval engine—SIREN. http://gbdi.icmc.usp.br/siren/ [25 July 2008].
18. Felipe Joaquim C, Traina AJM, Traina C Jr. Retrieval by content of medical images using texture for tissue identification. *Proceedings of the 16th IEEE Symposium on Computer-based Medical Systems* (*CBMS*), New York City, NY, 2003; 26–27.
19. Ye B, Peng J-X. Invariance analysis of improved Zernike moments. *Journal of Optics A*: *Pure and Applied Optics* 2002; **4**:606–614. DOI: 10.1088/1464-4258/4/6/304.
20. Swain MJ, Ballard DH. Color indexing. *International Journal of Computer Vision* 1991; **7**(1):11–32. DOI: 10.1007/BF00130487.
21. Tzanetakis G. Automatic musical genre classification of audio signals. *Proceedings of 2nd International Symposium on Music Information Retrieval* (*ISMIR*), Bloomington, IN, 2001; 205–210.
22. Ning H, Dannenberg RB, Tzanetakis G. Polyphonic audio matching and alignment for music retrieval. *Proceedings of IEEE Workshop on Applications of Signal Processing to Audio and Acoustics* (*WASPAA*), New Paltz, NY, 2003; 185–188.
23. GBDI-ICMC-USP. GBDI Arboretum Library. http://gbdi.icmc.usp.br/arboretum/ [25 July 2008].