UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
ISSN 0103-2569

Seamlessly Integrating Similarity Queries in SQL

Maria Camila N. Barioni
Humberto Razente
Agma J. M. Traina
Caetano Traina Jr.

Nº 277

RELATÓRIOS TÉCNICOS

São Carlos – SP
Jul./2006

Seamlessly Integrating Similarity Queries in SQL

Maria Camila N. Barioni, Humberto Razente, Agma J. M. Traina, and Caetano Traina Jr.

ICMC – Institute of Mathematics and Computer Sciences
USP – University of São Paulo at São Carlos
Avenida Trabalhador Saocarlense, 400
CEP 13560-970 – São Carlos – SP – Brazil
FAX: +55 16 3373 9751
{mcamila, hlr, agma, caetano}@icmc.usp.br

Abstract. Similarity search has received great attention on modern database applications involving complex objects, since the queries over such objects are seldom based on exact matches, but rather on some notion of similarity, specific to each domain. However, the SQL query language does not provide support for similarity queries. This paper proposes to include similarity queries to SQL, adding a powerful set of similarity operators. Two kinds of complex objects are considered: those monolithically stored as Binary Large Objects, and those stored as sets of attributes in a relation. Concerning monolithic objects, we regard specifically similarity search in image datasets. The paper also proposes ways to efficiently store and retrieve complex objects, showing a prototype developed to validate the concepts and the syntax presented.

1 Introduction

The data stored in the Database Management Systems (DBMS) are increasingly diversified. Nowadays, complex data types such as multimedia data (e.g. images, audio, video and long texts), geo-referenced information, time series, fingerprints, genomic data and protein sequences, among others, are stored in DBMS.

Searching in traditional DBMS relies on the total ordering property held by the numbers and small-text domains, which enables their comparison using the relational operators $<$, $\leq$, $=$, $\neq$, $>$ 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 operator ($=$) is also not meaningful, as two identical elements (e.g. two identical images) rarely occur. Fortunately, complex data domains often allow the definition of similarity relations among pairs of objects, which enables them to be queried by similarity. As similarity comparisons requires a way to quantify how similar each pair of objects are, complex data domains are often considered as metric spaces [1].

Formally, a metric space is a pair $< \mathbb{S}, d() >$, where $\mathbb{S}$ is the set of all objects 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 datasets with any L_p distance function, such as the Euclidean distance (L_2), are special cases of metric spaces. A distance function quantifies how similar two objects are, and it enables one to ask queries regarding the similarity of the objects, for instance "selecting the k objects most similar to a given one".

Complex objects can be stored in a database either as a set of traditional attributes or as a large binary object. 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 complex objects are called **particulate** objects, and the later **monolithic** objects. Particulate objects can be compared by similarity using a distance function defined over their attributes. For example, geo-referenced objects usually employ the Euclidean distance function over the attributes storing the coordinates of the object. Monolithic objects require extracting predefined features of them that are used in place of the objects to define the distance function. For example, 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) [2].

Often complex objects can be compared in different ways. For example, images can be compared by similarity of their color distribution, or by similarity of their texture. This can be accomplished by extracting several features and defining a specific distance function for each way in which the objects can be compared. Therefore, when comparing complex objects in a query, it is required to specify which distance function must be used in such query.

As objects of complex types are being stored at an increasing rate in relational databases, the need to support similarity queries also increases. However, the current standard of SQL does not consider similarity queries. This paper aims at providing this support, meeting the following requirements:

- How to represent powerful similarity queries, yet having a low impact over the existing SQL syntax;
- How to perform similarity queries over both monolithic and particulate data types, using the equivalent syntactical constructions of SQL;
- How to support the definition of any number of user-defined feature extractors (for monolithic objects) and adjustable metrics over complex data, granting them to every attribute defined in the same data domain.

Admittedly, it is conceivable that monolithic domains with particular properties can 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 type as a representative of the monolithic domains although many other monolithic domains could be equally supported.

Supporting similarity queries from inside SQL in a native form is important to provide optimization opportunities for the full set of search operations required to execute a query. While a procedural extension approach, such as employing user defined types and functions (UDT and UDF) from object-relational databases can use the existing highly optimized algorithms for each specific similarity operation, it does not allow optimizations among these operators nor their integration with the non-similarity based part of the query. Therefore, integrating similarity queries in a fully relational approach, as proposed in this paper, is a fundamental step to allow the supporting of complex objects as "first class citizens" in modern database management systems.

The remainder of this paper is structured as follows. Section 2 summarizes existing related works. Section 3 structures interesting concepts regarding types of similarity queries that are used in this work. The new syntax proposed to add similarity queries in SQL is presented in Section 4. Section 5 discusses the storage and execution of similarity queries. Finally, Section 6 gives the conclusions of this paper and suggestions of future works.

2 Related Work

Similarity is a fundamental paradigm to search large collections of complex data. Similarity queries search for objects close to a reference object according to a similarity measure, which is measured by a distance function. Actually a distance function gives the dissimilarity between two objects. The simplest queries use only one object from the data domain as the query reference. There are two main types of similarity predicates: the **Similarity Range query**, which retrieves the objects in the database that are dissimilar from the reference up to a given threshold; and the **k-Nearest Neighbor query**, which retrieves the k objects most similar to the reference. Queries that use a small set of objects as references have been called **Group Similarity queries** [3]. They have also been proposed for specific application domains [4]. Also, queries posed with no references rely upon the Cartesian product of two datasets, and are usually named as **Similarity Join queries** [5].

Recently, it has been achieved a great progress toward the enhancements of similarity queries. This involves some aspects as the increasing efficiency of query processing, for example with the development of compression techniques, dimensionality reduction techniques and index structures [6–10].

On the other hand, few proposals were aimed at extending SQL to allow similarity queries. A simple SQL extension was proposed in [11, 12]. These papers studied the addition of the `STOP AFTER` construction into the `SELECT` command as an optional suffix to enable declarative support for queries

involving ordering in the answers. Although these papers claimed to solve the emergent necessity of multimedia predicates, they do not show how this can be achieved.

Another extension, called SQL/sim [13], offers the ability of expressing the nearest neighbor queries in RDBMS through a user-defined predicate called NN-UDP. Although this proposal had pursued ways to support similarity queries in SQL, it is not able to provide a production-strength support seamlessly integrated with the other features of the SQL language. Moreover, it does not deal with all the predicates needed for the execution of similarity queries.

Table 1. Comparison of similarity query approaches.

Features allowed	SIREN	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 R2. ** DB2 AIV Extenders V. 8.

The International Standards Organization has also published the SQL/MM (SQL Multimedia and Application Packages, ISO/IEC 13249) [14], a multi-part standard proposal that provides storage and manipulation support for several multimedia data types based on user defined types and functions (UDT and UDF). While this extension can use the existing highly optimized algorithms for each specific similarity operation, it does not allow optimizations among these operators or their integration with the other operators used in a query. Commercial products such as Oracle InterMedia and IBM DB2 IAV Extenders follow this approach. Table 1 compares the main aspects of our proposed approach to representative available systems that allow posing similarity queries.

3 Fundamental Concepts

In this paper we consider that an object of a complex domain is either particulate or monolithic. Monolithic types are restricted herein to be only images, but the same concept could be extended to other complex data (e.g. audio and time series). Therefore, a relation can include any number of images or particulate attributes as its attributes. As an illustration, consider the `Employee` relation defined as:

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

Besides attributes of usual data types (e.g. `Name` and `HomeLat`), this relation has three attributes of complex data types: two `STILLIMAGE` and one `PARTICULATE`. An attribute of a complex type has one value in each tuple of the relation. This 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 dataset S, which composes a relation T. A similarity query is expressed using similarity predicates. A similarity predicate $P < d, lim, \mathbb{S} >$ uses a distance function d, defined over the complex domain $\mathbb{S}$, to rank the objects 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 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 operators always comes from one dataset $S \in \mathbb{S}$. The other operator can be either objects given as part of the predicate, or objects from another attribute getting its values from the same domain $\mathbb{S}$. The former are unary predicates, and the later are binary ones.

Unary predicates correspond to similarity selections. They compare the elements of one dataset $S \subset \mathbb{S}$ with one or more reference elements $s \in Q, Q \subset \mathbb{S}$ given as part of the predicate. A similarity selection over attribute S of relation T_1 can be represented as $\hat{\sigma}_{(S:P(d,Q,lim)}T_1$, where Q is a not-null dataset of query centers, and $S, Q \in \mathbb{S}$. The answer to a similarity selection is the subset of tuples from T_1, where the objects $s_i \in S$ remaining in the ranked list of the result of the predicate are stored.

When dataset Q has only one element s_q, the selection is the special case of the similarity range and k-nearest neighbor queries as follows.

- **Range selection** - *Rq*: given a maximum query distance ξ as the limit *lim*, the query $\hat{\sigma}_{(S:R_q(d,\{s_q\},\xi))}T_1$ retrieves every object $s_i \in S$ such that $d(s_i, s_q) \leq \xi$. An example is: "Select the Employees where their home coordinates are distant from position $< x, y >$ by at most 5 km, considering the Euclidean distance L_2", represented as $\hat{\sigma}_{(HomeCoordinate:R_q(L_2(),\{<x,y>\},5))}Employee$;
- k-**Nearest Neighbor selection** - kNN: given an integer value $k \geq 1$, the query $\hat{\sigma}_{(S:kNN(d,\{s_q\},k))}T_1$ retrieves the k objects that have the smallest distance from the query object s_q, according to the distance function $d()$. An example is: "Select the 3 frontal mugshots most similar to mugshot Im regarding texture", represented as $\hat{\sigma}_{(FrontalMugShot:kNN(Texture(),\{Im\}3))}Employee$.

When dataset 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 m_i from object s_i to the set of centers in Q. This measure is used by the similarity predicate to rank the objects in S. There are many ways to generate the aggregation. We consider in this paper that it is generated by the summation:

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

Different values for the power p can lead to different interpretations. For example, if $p = 1$, the resulting query will rank the objects s_i to minimize the summation of distances to the query centers, and if $p = 2$, it will rank the objects following their distances to the center of mass represented by the query centers.

Binary predicates correspond to similarity joins. They compare the elements of two datasets $S, R \in \mathbb{S}$ and rank pairs of objects $< s_i, r_j > | s_i \in S, r_i \in R$ regarding their distances. A similarity join over attribute S of relation T_1 and attribute R of relation T_2 can be represented as $T_1 \overset{S,R:P(d,lim)}{\bowtie} T_2$. The answer to a similarity join returns the concatenation of tuples, one from T_1 and the other from T_2, corresponding to the tuples where the values of s_i and r_j are stored. Similarity joins can also be limited either by distance ξ or the number of objects k, as follows.

- **Range Join** - $\overset{Rq}{\bowtie}$: given a maximum query distance ξ, the query $T_1 \overset{S,R:Rq(d,\xi)}{\bowtie} T_2$ retrieves the pairs $< s_i, r_j > | s_i \in S$, $r_i \in R$ such that $d(s_i, r_j) \leq \xi$.
- k-**Closest Neighbors Join** - $\overset{kCN}{\bowtie}$: given an integer value $k \geq 1$, the query $T_1 \overset{S,R:kCN(d,k)}{\bowtie} T_2$ retrieves the k closest pairs of object $< s_i, r_j > | s_i \in S, r_i \in R$.
- k-**Nearest Neighbors Join** - $\overset{kNN}{\bowtie}$: given an integer value $k \geq 1$, the query $T_1 \overset{S,R:kNN(d,k)}{\bowtie} T_2$ retrieves pairs of object $< s_i, r_j > | s_i \in S, r_i \in R$ such that there are k pairs for each object of S together with its nearest objects from R.

Every similarity operator allows a number of variations, such as retrieving the most dissimilar elements instead of the most similar ones, and taking into account occurrences of ties in k-limited predicates. There are also predicates limited by both k and ξ, and in this case the most restrictive condition overcomes the other.

4 Supporting Similarity Queries in SQL

To allow the processing of similarity queries on objects in complex domains, we must define the domains where similarity will be measured on new (complex) data types. We consider that the particulate and monolithic domains (restricted to images in this paper) must be defined as two new data types. In the following subsections we illustrate the use of SQL extended to support similarity queries. The structured description of the extension is available in the Appendix of this paper.

4.1 The CREATE METRIC Command

The first issue on including similarity queries in SQL is to allow the definition of distance functions. As there is no concept resembling the definition of comparison operators in SQL, it is needed to create new commands to do so. Distance functions are stored in the database catalog, thus their manipulation commands should follow the DDL command style. Hence, there will be three commands to handle distance functions: the CREATE METRIC, the ALTER METRIC and the DROP METRIC commands. We describe here only the CREATE METRIC, as the others follow similar constructions. Note that they are the only new commands needed to support similarity queries, as the other modifications are restricted to extensions on existing commands.

There are two variations of the CREATE METRIC command, one for PARTICULATE data types, and another for STILLIMAGE data types. The syntax of a CREATE METRIC for a PARTICULATE data type is:

```
CREATE METRIC <metric_name> USING <distance_function> FOR PARTICULATE
  (<place holder> <data type> [, <place holder> <data type>, ...]);
```

For example, the command to define a metric to evaluate the distance between two geographical points, represented by their Latitude and Longitude follows.

```
CREATE METRIC MyEuclidean2D USING LP2 FOR PARTICULATE
  (Latitude FLOAT, Longitude FLOAT);
```

When a metric is associated with a PARTICULATE data type, the place holders are associated with the attributes that compose the data type, as is shown in the next subsection. When a metric is associated with a STILLIMAGE data type, the place holders are associated with the features got by feature extractors. Feature extractors are defined as procedures that receive a complex object as an argument, and return any number of features, which is meaningful to compare complex data. The syntax of a CREATE METRIC for a STILLIMAGE data type is:

```
CREATE METRIC <metric_name> USING <distance_function> FOR PARTICULATE
  (<feature_extractor>(<feature> AS <place holder>,...),
  [<feature_extractor>(<feature> AS <place holder>,...), ...]);
```

Suppose the HistogramEXT extractor is defined elsewhere, which returns an array of integers representing the color histogram of one image. Then, the following command creates a metric called MyHistogram that can be used to compare images regarding color histogram.

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

4.2 Associating Metrics with Complex Data Types

Metrics can be associated with complex objects defined as attributes in any relation. The definition of how to compare pairs of values of complex types is expressed as a constraint for the attribute, following the two usual ways to define the constraints in a CREATE TABLE command: as a column constraint or as a table constraint. Moreover, as they enable creation of indexes to speedup queries, they can also be defined in a CREATE INDEX command.

Each monolithic object is stored as the value of an attribute having a monolithic type, such as the case of the STILLIMAGE data type. In the Employee relation, both FrontalMugShot and ProfileMugShot attributes are images, so they both can be compared using the MyHistogram metric. This can be defined as the column constraints:

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

Each object of a particulate domain is stored as a set of simple attributes in the relation. Therefore, to associate a metric with a particulate attribute, one must define which ones of the simple attributes that compose the complex attribute will be employed. For example, to use the `MyEuclidean2D` metric to compare `HomeCoordinate` values, the following table constraints can be used:

```
  ...
METRIC (HomeCoordinate)
  REFERENCES (HomeLat AS Latitude, HomeLongit AS Longitude)
  USING (MyEuclidean2D),
  ...
```

An equivalent column constraint could be used, provided that the composing attributes are defined prior to the particulate attribute in the `CREATE TABLE` command. The `REFERENCES` clause of the metric definition associates each attribute composing the particulate object with each place holder used in the metric. Monolithic domains do not need this clause because the associations are created in the feature extractor declaration in the corresponding metric. Several metrics can be associated with the same complex attribute, but in this case the word `DEFAULT` should be placed following one of the metric names.

Creating Indexes on Complex Data Types. It is also important to note that once a new feature extractor is included and defined as a new metric, it can be associated with any `STILLIMAGE` or `PARTICULATE` attribute of a relation already populated through a CREATE INDEX command. Index declarations enable the DBMS to use Metric Access Methods (MAM) to create one index for each metric associated with each attribute. The following are examples of creating indexes for the `HomeCoordinate` attribute and for the `FrontalMugShot` attribute.

```
CREATE INDEX Geography ON Employee(HomeCoordinate)
  REFERENCES (HomeLat AS Latitute, HomeLongit AS Longitude)
  USING MyEuclidean2D;

CREATE INDEX Frontal ON Employee (FrontalMugShot)
  USING MyHistogram;
```

4.3 Specifying Similarity Queries in the `SELECT` Command

The syntax of the DML commands (`SELECT`, `UPDATE` and `DELETE`) needs new constructions to allow expressing similarity predicates. In this paper we only describe the new constructions of the `SELECT` command, as the others are equivalent. The syntax of the `INSERT` command does not need changes.

The `SELECT` command allows a new construction for similarity predicates in the `WHERE` clause, and another to specify similarity joins in the `FROM` clause. The following subsections details each construction.

Similarity Conditions in the `WHERE` Clause. The new construction in the `WHERE` clause allows expressing all the similarity predicates described in section 3. The simplest expression of a non-similarity predicate compares an attribute with a constant value, in the format *attr* θ *value*, which expresses a selection. To express a similarity predicate, the attribute must be a complex one associated with a metric, the constant *value* must be an object of the corresponding complex domain, and the operator θ must be a similarity operator using a metric defined over the attribute. The basic syntax to express similarity selections is:

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

If the attribute is a STILLIMAGE, the constant image can be expressed as a path in the file system where the image is stored. For example, to select the five employees with frontal mugshots more similar to a given one, the following command can be used:

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

If the attribute is PARTICULATE, the constant value is expressed as a list of **<place-holder> AS <value>**. For example, to select the employees living nearer than 2 units from a given coordinate, the following command can be used:

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

Constant values can also be obtained from the database. Suppose we want to retrieve the five employees living nearer to the employee with **name='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;
```

The inner SELECT in the previous command can return more than one tuple if **name** is not a key. In this case, the command provides a set of centers to the query predicate, and an aggregation method should be chosen, following Equation 1 in Section 3. A keyword inserted after the keyword NEAR provides this choice: SUM for $p = 1$, ALL for $p = 2$ and MAX for $p = \infty$.

For example, to retrieve the name and the 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;
```

The usual construction to express a join operation in the WHERE clause is comparing an attribute from one table to an attribute from the other table, in the format $r.attr_1 \theta s.attr_2$. Similarity joins can be expressed using the same format, where both attributes are from the same complex domain, and both have been associated with the same metric. The construction $r.attr_1$ NEAR $s.attr_2$ RANGE ξ expresses a range join, the construction $r.attr_1$ NEAR $s.attr_2$ STOP AFTER k expresses a nearest join, and the construction $r.attr_1$ NEAR ANY $s.attr_2$ RANGE ξ expresses a closest join. For example, the following command retrieves the five pairs of employees whose frontal profiles look more similar to each other:

```
SELECT * FROM Employee E1, Employee E2
  WHERE E1.FrontalMugShot NEAR E2.FrontalMugShot
    STOP AFTER 5;
```

Variations on the basic command can be expressed with modifiers in the command. If one wants to retrieve the most dissimilar elements instead of the most similar, the word NEAR is replaced by the word FAR. If more than one metric was defined, the default one is used, unless the clause **BY <metric name>** is used. For example, suppose another metric, called MyTexture, was also associated with the attribute FrontalMugShot. Then, to select up to five employees with frontal mugshots more similar to a given one considering both MyHistogram and MyTexture metrics, the following command can be used:

```
SELECT * FROM Employee
  WHERE (FrontalMugShot NEAR 'c:/img.jpg' By MyHistogram
         STOP AFTER 5) AND
        (FrontalMugShot NEAR 'c:/img.jpg' By MyTexture
         STOP AFTER 5);
```

Queries limited to k neighbors (either in selections or joins) can take into account the occurrence of ties. The default behavior of a k-limited query is retrieving k elements without ties. However, it is possible to specify `WITH TIE LIST` following the `STOP AFTER` specification to include every object tied at the largest included distance.

Both `STOP AFTER` and `RANGE` can be specified in the same query. In this case, the answer is limited by having at most k objects and objects not farther (or nearer) than a 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 5 images not farther than 0.03 units from the given image.

Similarity Joins in the `FROM` Clause. An alternative syntax to express similarity joins exists for the `FROM` clause. This syntax, shown in Appendix A.5, follows the traditional representation of joins in that clause, using a syntax equivalent to the one in the `WHERE` clause.

Operator Precedence and Properties. Similarity predicates limited by ξ are commutative both for predicates limited by ξ as well as for non-similarity predicates. On the other hand, the predicates limited by k are not commutative with any kind of predicate. In fact, it is straightforward to verify that $\sigma_{(attr\ \theta\ value)}(\hat{\sigma}_{(S:kNN(d,\{s_q\},k))}T_1) \subseteq \hat{\sigma}_{(S:kNN(d,\{s_q\},k))}(\sigma_{(attr\ \theta\ value)}T_1)$. The non-commutativity of k-limited predicates precludes the equivalence of sequences of predicates to conjunctions of predicates and to intersections of the individual result sets. This is an undesired effect, as SQL expressions rely on the `AND` connector to represent sequences of predicates. To avoid ambiguity in evaluating similarity predicates, we established the following rules: 1) similarity predicates expressed in the `WHERE` clause are always executed before any non-similarity predicate; 2) k-limited predicates are executed before ξ-limited predicates; and 3) two or more k-limited predicates are executed independently over the base table and their results are met (if connected by `AND`) or joined (if connected by `OR`). Therefore the following two commands:

```
SELECT * FROM Employee
  WHERE FrontalMugShot NEAR 'c:/img.jpg' STOP AFTER 5
    AND HomeLat>23.2;
SELECT * FROM Employee
  WHERE HomeLat>23.2 AND
    FrontalMugShot NEAR 'c:/img.jpg' STOP AFTER 5;
```

are both translated into the same algebraic expression:

$$\sigma_{(HomeLat>23.2)}(\hat{\sigma}_{(FrontalMugShot:kNN(Metric(),\{\text{'c:/img.jpg'}\},5))}Employee).$$

Imposing those rules makes it easier for the query optimizer to use index structures to perform similarity searches, which is a desirable effect, as similarity search operations are usually more expensive than the traditional search operations. It also enables any non-k limited predicate to be handled in the usual way after processing the k-limited ones. However, a query requiring the opposite sequence cannot be expressed in a single SQL command using only the `WHERE` clause. Nevertheless, it can be expressed preparing the tables using the SQL:1999 nested table facility in the `FROM` clause.

5 Implementation

Besides proposing to add new constructions in the SQL language to represent similarity queries, this paper also deals with supporting queries over complex objects, which requires techniques to store the complex objects and algorithms to execute the queries. This Section tackles such requirements.

To evaluate the adequacy of the syntax proposed in this paper, we implemented SIREN, a *SImilarity Retrieval ENgine* that acts like a blade between a conventional DBMS and the application programs [15, 16]. SIREN intercepts every SQL command sent from the application. If it has no similarity construction nor a reference to complex objects, it sends the command to the underlying DBMS and sends back the answer from the DBMS to the application program. Therefore, when only conventional commands are posed by the application, SIREN is transparent. When the SQL command has similarity-related demands or references to complex objects, the command is re-written to execute the similarity-related operations internally using the underlying DBMS to execute the conventional data operations.

5.1 Storing Complex Objects for Similarity Retrieval

Particulate data types are stored by their constituent parts, presenting no new storage requirements. Monolithic data types are stored as Binary Large Objects (`BLOB` data types), but they also require storing their extracted features. Feature extraction is usually costly, and it must be executed for each object once, when the object is stored in the database. The features are stored as textual or numeric attributes associated with the complex object. As the user does not provide places in the relations to store the extracted attributes, the system must provide the places to store the extracted features and their association with the `BLOB` data in a way transparent to the user.

To store the features extracted from a monolithic object, such as a `STILLIMAGE` object, SIREN changes the definition of user defined tables that have monolithic attributes as follows. Each `STILLIMAGE` attribute is changed to a reference to a system-controlled table that has as its attributes both the `BLOB` and a set of attributes that stores all features got by every extractor used in each metric associated with the attribute. A new table is created for each `STILLIMAGE` attribute. Whenever a new image is stored in the database, SIREN intercepts the `INSERT` command, stores the non-image attributes in the user table and the images in the corresponding system tables. Then, SIREN calls the feature extractors and stores their outputs in the corresponding system tables. Whenever the user asks for data from its tables, SIREN joins the system tables and the user tables, removing the feature attributes, so that the user never sees the table split nor the features. Figure 1 shows an illustration of how these new data types are stored by SIREN.

When the user poses queries involving similarity predicates, SIREN uses the extracted features to execute the similarity operators. The current version of SIREN supports `PARTICULATE`, `STILLIMAGE` and `AUDIO` attributes It has three types of feature extractors regarding the `STILLIMAGE` data type: a texture extractor (`TEXTUREEXT`), a shape extractor based on Zernike Moments (`ZERNIKEEXT`) and a color extractor based on the normalized color histogram (`HISTOGRAMEXT`) [17]. And there is only one extractor for the `AUDIO` data type: a sound texture extractor (`SOUNDTEXTUREEXT`) that extracts features based on Short Time Fourier Transform (STFT) and Mel-Frequency Cepstral Coefficients (MFCC) [18, 19]. The MAM employed by SIREN to index PARTICULATE and STILLIMAGE attributes is the Slim-Tree [8], available in the Arboretum (an open source C++ library which implements various MAMs) [20]. We assure that an eventual implementation of a DBMS with native support for similarity queries can follow this same approach.

5.2 Similarity Queries in SIREN

The similarity operators implemented to execute the similarity queries consist of two similarity selections for a single query center, three for multiple centers and three similarity joins. The first two operators correspond to the similarity range and k-nearest neighbor ones already developed in Slim-tree, but there is no procedure published to execute the other operators in this or in any other MAM. Therefore, we implemented the remaining six procedures using sequential scan to validate SIREN and the proposed approach to support similarity queries in SQL. If a complex attribute is associated with

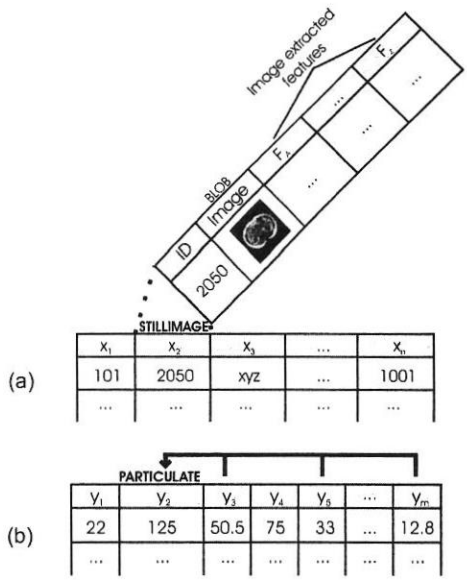

Fig. 1. Storage schema of the new complex data types. (a) `STILLIMAGE` data type. (b) `PARTICULATE` data type.

an index, SIREN executes single center similarity selection using a Slim-tree. If the attribute does not have an index, or the required operator is not a single center selection, then the query is answered using sequential scan.

In order to illustrate the new data domains defined, two data sets are employed. The first one, called `MedImages`, is composed of 5,180 medical images. The images are Computerized Tomographies (CT) from three human body parts: abdomen, cranium and thorax. Each tuple of this data set has an image id, the image, the description of the body part and an attribute that specifies whether the image identifies a pathological condition. Similarity queries can be posed to explore this data set considering several aspects of the images such as the similarity based on color distribution or on texture. Two metrics were defined to handle these features, one using the `HISTOGRAMEXT` extractor and other using the `TEXTUREEXT` extractor. The commands employed to create the table and the metrics follows.

```
CREATE METRIC HistogramLP1 USING LP1 FOR STILLIMAGE
       (HISTOGRAMEXT (Histogram AS Histo));

CREATE METRIC Texture FOR STILLIMAGE
       (TEXTUREEXT (Texture AS T));

CREATE TABLE MedImages (
       Id INTEGER PRIMARY KEY,
       BodyPart VARCHAR(15),
       Img STILLIMAGE,
       Pathology CHAR(1),
       METRIC (Img) USING
         (HistogramLP1 DEFAULT, Texture));
```

Examples of queries that utilize each one of these metrics are shown in Figure 2.

The second data set, called `Cars`[1], is composed of the description of 392 cars. This data set is composed of five attributes that describe the following variables: MPG (miles per gallon), horsepower, time to accelerate from 0 to 60 mph (sec.), origin of car (American, European or Japanese) and the

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

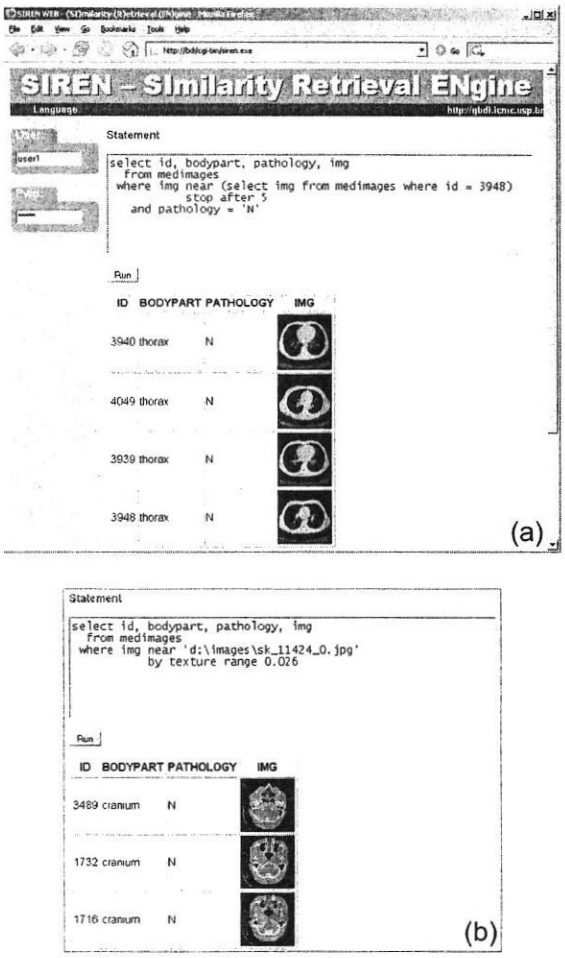

Fig. 2. Similarity query examples based on MedImages table. (a) *k-NN query* considering the default metric (`Histogram`). (b) *Range query* considering the `Texture` metric.

car names. This data set can be queried by similarity considering several questions. Here we show a metric to compare cars based on the cost-benefit ratio related to the variables horsepower, acceleration and MPG. The commands employed to create the table and the metric described above are presented 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));
```

Several metrics can be associated with each complex attribute. Using the metric associated with the Cars data set, it is possible to perform queries such as those presented in Figure 3.

(a)

Statement

```
select carname, horsepower, consumption, acceleration, origin
  from cars
 where car near (67 as hp,
                 38 as mpg,
                 15 as sec)
               stop after 3
```

Run

CARNAME	HORSEPOWER	CONSUMPTION	ACCELERATION	ORIGIN
ford fiesta	66	36.1	14.4	American
honda civic	67	38	15	Japanese
datsun 310 gx	67	38	16.2	Japanese

(b)

Statement

```
select carname, horsepower, consumption, acceleration, origin
  from cars
 where car near (select horsepower as hp, consumption as mpg,
                       acceleration as sec
                  from cars
                 where carname = 'ford mustang')
               stop after 10
   and origin <> 'American'
```

Run

CARNAME	HORSEPOWER	CONSUMPTION	ACCELERATION	ORIGIN
audi 100 ls	90	24	14.5	European
mazda rx3	90	18	13.5	Japanese
audi 100ls	91	20	14	European

(c)

Statement

```
select americancars.carname,
       europeancars.carname
  from americancars, europeancars
 where americancars.car near europeancars.car
       stop after 3
```

Run

CARNAME	CARNAME_1
chevrolet chevelle malibu	mercedes-benz 280s
chevrolet chevelle malibu	volvo 264gl
chevrolet chevelle malibu	peugeot 604sl
buick skylark 320	mercedes-benz 280s
buick skylark 320	volvo 264gl
buick skylark 320	peugeot 604sl

Fig. 3. Similarity query examples based on the Cars table. (a) "Which are the 3 most similar cars having: Horsepower = 67 hp, Consumption = 38 mpg and Acceleration = 15 s?". (b) "Which are the 10 cars most similar to the given car and whose origin is not American?". (c) "Which are the 3 most similar European cars to each American car?".

Considering the execution of a query, the time spent by the search in a MAM depends on several factors such as: the size of the feature vector; the number of objects indexed; and the time needed to read a block in a hard disk. As a reference, the total time spent by SIREN to analyze and execute the query presented in Figure 2(b) was 0.29 seconds and in Figure 3(b) was 0.16 seconds, running in a 1.8 GHz PC (including the time spent to read the data dictionary, but not the time spent to generate the image thumbnails and the preparation of the html page).

6 Conclusions

Searching large sets of complex objects by similarity has attracted much attention in the database and machine learning communities. However, each work being developed until now focuses on specific topics

related to the broad concept of similarity, but there is no conducting line that is able to join together these many disperse efforts. Supporting similarity queries in SQL, enabling the seamless integration of similarity queries with the other resources of the language, is a strong reason to be that conducting line. This work presents such support.

The presented support for similarity queries is powerful enough to allow several variations, including selections and joins by similarity, as well as similarity comparisons involving groups of objects. Every kind of query 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 objects stored as a particulate set of composing attributes. Moreover, the support also enables the association of feature extractors of complex objects with the metrics, making it possible to support content-based retrieval of complex objects. In this paper, we have shown specifically how to perform content-based image retrieval as a representative example of monolithic objects, besides the particulate objects. We also report results of using a prototype that implements similarity queries on images and particulate data using an Oracle database server.

As a conducting line to join together research activities, our solution for the similarity query representation has also 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 target at similarity, such as the development of index structures that support similarity queries. This characteristic can also guide the development of such structures, as it establishes the types of retrieval operations that are worth improving. 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 data structure) can be an interesting possibility in the future.

Acknowledgments

This work has been supported by FAPESP (São Paulo State Research Foundation), CNPq (Brazilian National Council for Supporting Research), and CAPES (Brazilian Coordination for Improvement of Higher Level Personnel).

References

1. Ciaccia, P., Patella, M.: Searching in metric spaces with user-defined and approximate distances. ACM Transactions on Database Systems (TODS) **27** (2002) 398–437
2. Smeulders, A.W.M., 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 **22** (2000) 1349–1380
3. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.K.: Aggregate nearest neighbor queries in spatial databases. ACM Transactions on Database Systems **30** (2005) 529–576
4. Papadias, D., Shen, Q., Tao, Y., Mouratidis, K.: Group nearest neighbor queries. In: IEEE International Conference on Data Engineering (ICDE), Boston, MA (2004) 301–312
5. Böhm, C., Krebs, F.: High performance data mining using the nearest neighbor join. In: IEEE International Conference on Data Mining (ICDM), Maebashi City, Japan (2002) 43–50
6. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: International Conference on Very Large Data Bases (VLDB), Athens, Greece, Morgan Kaufmann (1997) 426–435
7. Balko, S., Schmitt, I., Saake, G.: Towards enhanced compression techniques for efficient high-dimensional similarity search in multimedia databases. In: EDBT Workshops 2002. (2002) 365–375
8. Jr., C.T., Traina, A.J.M., Faloutsos, C., Seeger, B.: Fast indexing and visualization of metric datasets using slim-trees. IEEE Transactions on Knowledge and Data Engineering (TKDE) **14** (2002) 244–260
9. Digout, C., Nascimento, M., Coman, A.: Similarity search and dimensionality reduction: Not all dimensions are equally useful. In: International Conference on Database Systems for Advanced Applications (DASFAA). Volume 2973., Springer LNCS (2004) 831–842

10. Kailing, K., Kriegel, H.P., Schönauer, S., Seidl, T.: Efficient similarity search for hierarchical data in large databases. In: International Conference on Extending Database Technology (EDBT). Volume 2992 of Springer LNCS. (2004) 676–693
11. Carey, M., Kossmann, D.: On saying "enough already!" in sql. In: SIGMOD 1997. (1997) 219–230
12. Carey, M., Kossmann, D.: Reducing the braking distance of an sql query engine. In: International Conference on Very Large Data Bases (VLDB), New York City, USA, VLDB Endowment, Saratoga, Calif. (1998) 158–169
13. Gao, L., Wang, M., S.W., X., Padmanabhan, S.: Expressing and optimizing similarity-based queries in sql. In: International Conference on Conceptual Modeling (ER). Volume 3288 of Springer LNCS. (2004) 464–478
14. Melton, J., Eisenberg, A.: SQL Multimedia and Application Packages (SQL/MM). SIGMOD Record **30** (2001) 97–102
15. GBDI-ICMC-USP: Similarity retrieval engine – SIREN. http://gbdi.icmc.usp.br/siren/ (2006)
16. Barioni, M.C.N., Razente, H., Traina, A.J.M., Traina-Jr, C.: SIREN: A similarity retrieval engine for complex data. In: International Conference on Very Large Data Bases (VLDB), 4 p, To be published (2006)
17. Long, F., Zhang, H., Feng, D.D.: Fundamentals of content-based image retrieval. Multimedia Information Retrieval and Management, Springer (2003)
18. Tzanetakis, G.: Automatic musical genre classification of audio signals. In: International Symposium on Music Information Retrieval (ISMIR). (2001) 205–210
19. Hu, N., Dannenberg, R.B., Tzanetakis, G.: Polyphonic audio matching and alignment for music retrieval. In: IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA). (2003) 185–188
20. GBDI-ICMC-USP: Gbdi arboretum library. http://gbdi. icmc.usp.br/arboretum/ (2006)

Appendix

A Syntax for Similarity Queries in SQL

In this Appendix we present the detailed syntax of similarity query support in SQL. The use of [ ] means a choice of optional terms, and { } means a choice of required terms. Regarding the DDL commands, we present here only the syntax of the `CREATE` ones, as the syntax of the corresponding `ALTER` and `DROP` commands follows the same structure.

A.1 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>')'}

<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>]
```

A.2 Specifying METRIC as a Column Constraint

The METRIC constraint can be associated with STILLIMAGE or PARTICULATE data types. 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>
      | <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 only to define a column constraint for PARTICULATE attributes.

Attributes of a complex data type not associated with a metric cannot participate in a similarity predicate. If a complex attribute is associated with two or more metrics, then a default metric should be specified. One metric can be associated with any number of complex attributes, but both the attributes and the metric must be defined over the same complex data type, that is either STILLIMAGE or PARTICULATE.

A.3 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>')'
       | '('<particulate_attr_name_list>')'
         REFERENCES '('<param_assoc_list>')'
       } USING '('<metric_name_list>')'
```

A.4 CREATE INDEX for Metric Indexes

There is one variation of the CREATE INDEX command specific for STILLIMAGE attributes, and one for the PARTICULATE ones. These variations are expressed as:

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

A.5 Specifying Similarity Queries in the SELECT Command

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

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

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

<similarity_grouping>::= SUM | ALL | MAX
```

Attribute <complex_attr_name$_1$> is the one to be searched in the comparison predicates. It can be compared to a constant value <attr_value>, with a set of constant values '('<attr_value_set>')' or with another attribute in any table of the database. If attribute <complex_attr_name$_1$> is a STILL-IMAGE and must be compared to a constant value, the constant is an image expressed as a path in the file system where the image 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 <complex_attr_name$_1$> to a constant or to a set of constants represents a similarity selection. Comparing the attribute to another attribute in the same domain represent a similarity join. Specifying both STOP AFTER and RANGE clauses requires considering both k and ξ limits.

If neither STOP AFTER nor RANGE is specified then RANGE 0 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 a select query. If the <similarity_grouping> 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.

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.

Other Constructions. The features extracted from monolithic domains can be used in SQL commands whenever an attribute reference is allowed, using the following syntax;

<attr_reference>::= [[<rel_name>'.']<attr_name>'.']
<extractor_name>'.'<param_alias>

The distance between the query center and the object in the query answer can be returned in similarity selection queries, provided there is only one similarity predicate in the query. This value can be requested in SQL commands whenever an attribute reference is allowed, using the following syntax.

<attr_reference>::= <extractor_name>()

For a similarity selection with more than one query center, this construction returns the aggregate distance used to choose each tuple, that is: the summation of distances of the object to the set of centers for SUM; the squared root of the summation of these distance squared for ALL; and the least distance of the distances of the object for an object in the input group for MAX. For a similarity join, this construction returns the distance between each object in the returned pair.