FMI-SiR: A Flexible and Efficient Module for Similarity Searching on Oracle Database

Daniel S. Kaster[1,2,*], Pedro H. Bugatti[2], Agma J. M. Traina[2] and Caetano Traina Jr.[2]

[1] Department of Computer Science, University of Londrina, Londrina, PR, Brazil
dskaster@uel.br
[2] Department of Computer Science, University of São Paulo at São Carlos, SP, Brazil
{pbugatti, agma, caetano}@icmc.usp.br

Abstract. The volume of complex data (images, videos, audio, time series, DNA sequences, and others) has been growing at a very fast pace. Although they are not naturally handled by Database Management Systems (DBMSs), it is necessary to store them in databases. Complex data are well-suited to be queried by similarity, and several works have been addressing techniques for similarity searching. However, the majority of the techniques is not conceived to be integrated into a database engine. To include similarity search into the database core requires allow taking advantage of the DBMS resources to perform queries, integrating complex and conventional data. Oracle Corp. developed the Oracle interMedia module to support multimedia data in its database manager, providing several operations to manipulate them. It allows performing content-based image retrieval through proprietary functions to extract intrinsic features from images and to compute their similarity. In this paper we describe another module for similarity search, also developed using the Oracle's Extensible Architecture Framework. Our approach allows including user-defined feature extraction methods and distance functions into the database core, whereas providing wider flexibility. The similarity operators supported include both similarity selection on a single relation, as well as similarity range joins performed over two relations. The experiments show that employing our module to query images by content improves the results obtained using Oracle alone, both in the precision of the results and in the performance of executing queries.

Categories and Subject Descriptors: H. Information Systems [**H.m. Miscellaneous**]: Databases; H. Information Systems [**H.3. Information Storage and Retrieval**]: H.3.1. Content-Analysis and Indexing

Keywords: Content-Based Image Retrieval, Metric Access Methods, Similarity Queries

1. INTRODUCTION

Database Management Systems (DBMSs) provide effective resources to perform queries comparing data using the relationships of equality and order. However, complex data, such as audio, video, time series and DNA sequences, cannot be compared by such criteria, as ordering does not apply, and it is very unlikely to have two complex elements exactly equal. Therefore, similarity is the most rewarding concept employed to retrieve complex data by content.

Similarity comparison requires a distance (or dissimilarity) function to quantify how close (i.e. dissimilar) each pair of data elements is. In order to enable DBMSs to provide support for content-based similarity comparison of complex data, it is necessary to implement algorithms to extract intrinsic characteristics (features) from the data, generating mathematical signatures that represent them. The distance functions employed to compare complex data are usually based on their signatures. Attaching feature extraction algorithms and distance functions into the database core is necessary and sufficient to allow DBMSs to integrate complex and conventional data querying.

In [Kaster et al. 2009] we described a module called the FMI-SiR (user-defined **F**eatures, **M**etrics

*Corresponding author. On leave at Dept. of Computer Science, University of São Paulo at São Carlos, SP, Brazil.

and **I**ndexes for **Si**milarity **R**etrieval), developed to perform data selection on a relation executing similarity search operations. It included the two required mechanisms aforementioned into a database core to perform similarity search, as well as included Metric Access Methods (MAMs) to efficiently execute similarity queries based on selections. To evaluate our approach, we implemented a version of FMI-SiR using the Oracle's Extensible Architecture Framework to handle complex data, called the FMI-SiR$_O$. Applications employing complex data can take advantage of FMI-SiR$_O$, by defining its own domain specific feature extractors and/or distance functions, whereas using the powerful resources of the Oracle database in an integrated way.

In this paper we present further research accomplished on the FMI-SiR. We highlight that FMI-SiR is also capable to perform indexed range join operations on a pair of relations sharing atributes from a single complex domain. Thus, since the original FMI-SiR publication in [Kaster et al. 2009], we worked on several new developments and experiments, more specifically regarding range join operations and also its scalability.

The remainder of this paper is structured as follows. Section 2 summarizes the concepts needed to understand our approach and existing related works. Section 3 presents the FMI-SiR$_O$ module, while Section 4 presents experiments performing join operations and discusses the results achieved. Finally, Section 5 presents the conclusions and future directions.

2. BACKGROUND AND RELATED WORK

This section presents the main concepts to understand the techniques required to perform similarity comparison in complex data domains. It also presents existing approaches to execute similarity queries on DBMSs.

2.1 Similarity Search and Metric Spaces

In this paper, a complex data domain is called the one whose requirements to perform similarity comparison are fulfilled, which means that it has a function defined to measure similarity. It is often defined as a distance function, which quantifies how dissimilar two elements are. A similarity query returns the stored elements that satisfy a given similarity criterion.

Similarity selections are usually expressed in terms of one or more reference elements, which are called the *query center(s)*. The two main comparison operators employed to perform similarity selections on a relation having a complex attribute follows:

Range query (Rq). given a dataset $S \in \mathbb{S}$, a distance function δ, a query center $s_q \in \mathbb{S}$ and a radius $\xi \in \mathbb{R}^+$, a range query selects every element $s_i \in S$ such that $\delta(s_i, s_q) \leq \xi$;

k-Nearest Neighbor query (k-NNq). given a dataset $S \in \mathbb{S}$, a distance function δ, a query center $s_q \in \mathbb{S}$ and an integer value $k \geq 1$, the nearest neighbor query selects the k elements $s_i \in S$ that have the shortest distance from s_q.

Similarity joins [Böhm and Krebs 2004] are useful to cross information of two datasets. One of the most employed similarity join is the Range join, as follows.

Range Join (Rj). Given two datasets $R, S \in \mathbb{S}$, a distance function δ and a radius $\xi \in \mathbb{R}^+$, a range join retrieves the pairs $\langle r, s \rangle$, $r \in R$ and $s \in S$, such that $\delta(s, r) \leq \xi$;

The nature of similarity queries precludes using conventional indexes based on the total order property to speed up their execution. Many complex data can be represented as points in a vector or in a multidimensional space, but there also are "adimensional" domains, such as words and genetic

sequences. Metric spaces are adequate to represent complex data, as they only require the elements and their pairwise distance, being data either dimensional or adimensional [Zezula et al. 2006].

A metric space is formally defined as a pair $\mathbb{M} = \langle \mathbb{S}, \delta \rangle$, where $\mathbb{S}$ denotes the universe of valid elements and δ is a distance function $\delta : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$ (called a metric) that satisfies, for any $s_1, s_2, s_3 \in \mathbb{S}$, the following properties: (1) Symmetry: $\delta(s_1, s_2) = \delta(s_2, s_1)$; (2) Non-negativity: $0 < \delta(s_1, s_2) < \infty$ if $s_1 \neq s_2$ and $\delta(s_1, s_1) = 0$; and (3) Triangular inequality: $\delta(s_1, s_3) \leq \delta(s_1, s_2) + \delta(s_2, s_3)$.

There are several kinds of metrics, however the most employed are those from the Minkowski family [Long et al. 2003]. Nonetheless, it has been shown that there is an appropriate relationship between the intrinsic features extracted from complex data and the distance function employed that improve the quality of the similarity evaluation [Bugatti et al. 2008].

Metric Access Methods (MAMs) are structures developed to index data in metric domains. Existing MAMs can be classified as: (i) static, which are constructed in a single operation using the whole dataset and need to be rebuilt upon modifications, such as the BK-Tree [Burkhard and Keller 1973] and the VP-Tree [Yianilos 1993]; and (ii) dynamic, which allow further modifications, such as the M-tree [Ciaccia et al. 1997; Skopal and Lokoč 2009], the BM$^+$-tree [Zhou et al. 2005], the DBM-tree [Vieira et al. 2004] and the Slim-tree [Traina Jr. et al. 2002].

Indexing structures are usually developed under a software library that provides the primitives required to their execution. The similarity querying module that is described in this paper attaches the Arboretum library to the Oracle database, providing indexed search. Therefore, the next subsection introduces this library.

2.2 The Arboretum Library

Arboretum[1] is a C++ open source software library, developed by the Databases and Images Group at ICMC–USP, which implements several indexing structures, mainly MAMs. It is organized in 3 different and independent layers. The "User layer" defines the application-dependent object types and the distance functions employed to compare them. Indexing structures are defined in the "Structure layer". It provides the methods to manipulate the index, including object insertion, query answering and structure optimizations. The "Storage layer" contains the page managers, which manage storing pages into memory or disk.

There are many indexing structures implemented on the Arboretum. We use the Slim-tree to develop the FMI-SiR$_O$ module, as it is a mature, fast and scalable structure. In the next subsection we briefly describe the Slim-tree.

2.3 The MAM Slim-tree

Slim-tree is a dynamic, height-balanced and bottom-up constructed tree, presenting the first technique to measure and reduce overlap between subtrees. It divides the metric space into not necessarily disjunct regions, each one composing a tree node. Each node is stored as a fixed size disk page, and has one element acting as a representative for the elements stored at that node and a node covering radius. A node defines a region of the metric space whose the center is the representative element, that covers all elements in this node, or in any subtree rooted at one of its elements. Slim-tree has two node types: the leaf nodes, whose entries are the indexed elements, and the index nodes, where the representatives of the subtrees are replicated in the hierarchy.

The representatives guide the search that answers a query, enabling pruning the subtrees that do not contain result candidates, as well as reducing the amount of distance calculations, since the distances between the elements and their representatives are pre-computed during insertion.

[1]Available at http://gbdi.icmc.usp.br/arboretum

2.4 Support for Similarity Queries in DBMSs

Similarity applies to a broad variety of data domains. However, to support similarity queries in a DBMS it is necessary to provide the distance functions and the feature extractors specific to each application domain. For example, many systems requires retrieving digital images by content, such as medical and agrometeorological applications. This section presents existing solutions to enable a DBMS to support similarity searching over a set of images.

The ISO/IEC SQL/MM is a standard proposed for multimedia data. Its SQL/MM StillImage is devoted to images, defining an abstract data type that encapsulates the binary image, its metadata, a set of basic image processing functions and data types aimed at executing similarity queries over image contents. However, this standard does not address either query representation or similarity query processing. Thus, each DBMS vendor uses its proprietary extensions.

The IBM DB2 was one of the first DBMS to offer a module to query images by content, named the DB2 Image Extender [IBM Corporation 2003]. The query-by-content functionality is based on the QBIC system [Flickner et al. 1995], and both Rq and k-NNq are provided. Modules from other database vendors, such as the Informix Excalibur Image Data Blade [Informix Corporation 1999] and the Oracle interMedia [Oracle Corporation 2005] also provide support for image query-by-content. However, these modules are commercial and their source code are not available to make improvements and develop domain specific features. There are a variety of image contents and although the fact that they could be used for different purposes, the generalist approaches provided by these systems are not satisfatory for many applications.

There are academic open source systems to perform similarity search over images, such as SIREN [Barioni et al. 2005] and PostgreSQL-IE [Guliato et al. 2008]. SIREN (SImilarity Retrieval ENgine) is a prototype that implements an interpreter over Oracle and PostgreSQL. It recognizes an extension to the SQL language [Barioni et al. 2009] that allows the representation of similarity queries. PostgreSQL-IE (PostgreSQL with Image-handling Extension) is a prototype extension to PostgreSQL that encapsulates the images in a new data type and provides a number of feature extractors for medical images. Nevertheless, the similarity search on these systems are not fully integrated into the query plan, preventing exploiting optimization alternatives.

The FMI-SiR_O module presented in this paper is not only open source, enabling the inclusion of application specific features, but it can also be controlled by the DBMS query processor, allowing a tight integration to other DBMS operations. As it is implemented on the Oracle database, the next section explains the mechanism available on Oracle interMedia, to serve as a baseline to evaluate the developed module.

2.5 Image Similarity Search on Oracle interMedia

The Oracle interMedia [Oracle Corporation 2005] (now called Oracle Multimedia) is a module developed by Oracle Corp. to support multimedia data inside its DBMS. Content-based image retrieval is implemented through proprietary functions to extract features and generate signatures, which are encapsulated using the `ORDImageSignature` type. The most important functions regarding this type are `ORDImageSignature.generateSignature`, which generates a signature according to an input image, and `ORDImageSignature.evaluateScore`, which receives two image signatures and a string defining the parameters of similarity comparison, and returns the distance between these signatures. The similarity comparison is computed by a linear combination of the partial similarity of the descriptors: color, texture, shape and location.

Oracle interMedia allows to perform range and k-NN queries over images using the standard SQL syntax. The examples presented in this subsection consider the attribute `image` as an `ORDImage` type, and the image signatures are stored in the `ORDImageSignature` attribute `image_signature` of the

table `person`. The following query illustrates how a range query can be requested[2]:

```
SELECT * FROM person p
WHERE ORDImageSignature.evaluateScore(image_signature,
  ORDImageSignature.generateSignature('example.jpg'),
  'color="1.0" texture="0.8" shape="0.1" location="0"'
) <= 1.5;
```

Aimed at accelerating range queries, Oracle interMedia provides a specific operator, the `IMGSimilar`, which encapsulates a range query, allowing it to be indexed using a specific index type called the `ORDImageIndex`. The `ORDImageIndex` comprises a set of 63 bitmaps, filtering and reducing similarity calculations. The following statement creates an index of type `ORDImageIndex` to accelerate range queries:

```
CREATE INDEX image_ix ON person(image)
INDEXTYPE IS ORDImageIndex;
```

To perform an indexed range search, the query must be rewritten substituting the `ORDImageSignature.evaluateScore` by the `IMGSimilar` operator, which integrates the range search into the query plan.

A k-NN query can be solved using the window functions that are included in the SQL [ISO/IEC 2003] standard. Consider a k-NN query standard as follows:

```
SELECT * FROM (
  SELECT p.id, p.name, ROW_NUMBER() OVER (
      ORDER BY ORDImageSignature.evaluateScore(
        image_signature,
        ORDImageSignature.generateSignature('example.jpg'),
        'color="1.0" texture="0" shape="0.1" location="0"')
    ) AS rownum
  FROM person p
) WHERE rownum <= 10;
```

where the function `ROW_NUMBER` assigns the position of each tuple to the defined window. In this case the tuples are ordered following the similarity between the stored and the query images. Thus, this information allows returning the k-nearest neighbors. This type of query is always performed in Oracle by a sequential scan over the table.

Despite the tools that Oracle interMedia offers, only sequential scan is available to solve k-NN queries. Moreover, experimental evaluations revealed that Oracle's indexed range scan produces false negatives, i.e. it discards tuples that should be part of the result, making the indexed scan inconsistent with the sequential one. The next section describes the new FMI-SiR$_O$ module for similarity searching that we developed, which does not present such problems and, moreover, as it is open source, it can be enriched with new domain specific functionalities.

3. THE FMI-SIR$_O$ MODULE

This section describes the FMI-SiR$_O$ module, a FMI-SiR (user-defined **F**eatures, **M**etrics and **I**ndexes for **Si**milarity **R**etrieval) implementation over the Oracle DBMS. To explain its implementation, a running example is provided considering that a table called `image_table` exists, containing an identifier `id`, a BLOB attribute `image` populated with image data, and a BLOB attribute `image_sign` to store the signature that describes the image's content, which is used in the similarity queries.

[2]The queries presented in this section employ a slightly simplified syntax to improve readability.

3.1 Signature Generation and Sequential Similarity Querying

The first task addressed by the FMI-SiR$_O$ is to generate signatures describing the complex data content. To generate signatures we developed the `generateSignature` function, which can be called in the following way:

```
FOR csr IN (SELECT image, image_sign FROM image_table FOR UPDATE) LOOP
  generateSignature('', csr.image, csr.image_sign, 'Histogram');
END LOOP;
```

The SQL function `generateSignature` maps to a C++ function, whose execution is illustrated in Figure 1. When this function is called, the query processor provides a text parameter (detailed below), the feature extractor identification and the LobLocators of the BLOB attributes, which point to the stored raw binary data (Step 1). If the first parameter is empty, the function loads the image and the signature binary data pointed by the locators (Step 2). The image is forwarded to the feature extractor defined in the last parameter (in the example, the `Histogram` image color extractor), which processes the image and returns the feature vector (Step 3). The feature vector is serialized and stored in the attribute pointed by the locator `image_sign` (Step 4) and the function returns to the query processor (Step 5). Notice that the signature is an IN/OUT parameter, thus it is necessary to select the data using an exclusive lock, expressed in the example by the `FOR UPDATE` clause.

In this description of the FMI-SiR$_O$ implementation we address only image extractors. However, the module architecture supports including additional feature extractors and also loading features extracted by an external software stored in a text file. This functionality is enabled filling the first parameter of the `generateSignature` function with the feature file name. In this case, the last two parameters are ignored and the signature BLOB is populated with the features read from the file.

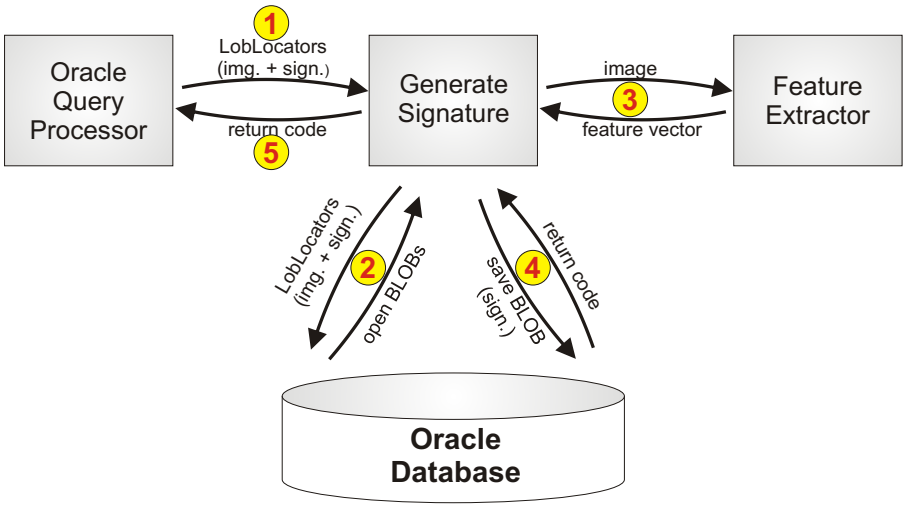

Fig. 1. Steps of function `generateSignature`.

After having the signatures computed, the complex data are ready to be compared by similarity. Similarity queries are based on similarity functions, which are defined in FMI-SiR$_O$ as follows:

```
<distance_name>_distance(signature1 BLOB, signature2 BLOB);
```

This command returns the distance between the two signatures as a real value. Several distance functions are available, such as the `Manhattan_distance`, the `Euclidean_distance` and the `Canberra_distance`. However, new distance functions can also be easily included according to application

requirements. These functions can be employed to formulate similarity queries following the standard SQL syntax, like those presented in Subsection 2.5. However, the queries are executed through a linear scan. To enhance performance, we developed indexes for similarity search, as described in the next subsection.

3.2 Similarity Index Definition and Construction

We developed a new index type for the Oracle query processor aimed at similarity searching, based on the Oracle's Extensible Indexing Interface. This interface allows defining the methods required by a data type implementing the new index type. The type that was created has the header as follows:

```
CREATE OR REPLACE TYPE index_im_type AS OBJECT (
  scanctx RAW(4),
  STATIC FUNCTION ODCIIndexCreate(),
  STATIC FUNCTION ODCIIndexDrop(),
  STATIC FUNCTION ODCIIndexInsert(),
  STATIC FUNCTION ODCIIndexDelete(),
  STATIC FUNCTION ODCIIndexUpdate(),
  STATIC FUNCTION ODCIIndexStart(),
  MEMBER FUNCTION ODCIIndexFetch(),
  MEMBER FUNCTION ODCIIndexClose());
```

The type definition body maps the functions to external C++ functions. A new index type was defined using this structure based on the Slim-tree implementation in the Arboretum library. In fact, we developed a number of new index types, one for each distance function available, as a Slim-tree is tightly coupled to the distance function employed to index the data. A complex data signature attribute can be used in similarity queries through distinct distance functions, but each index can only be used in queries that employ the same distance function employed for its construction. Thus, the index types are distinguished by the distance function employed. This approach avoids that the query processor use an invalid index and enables to create more than one index over the same complex data signature attribute for different distance functions.

Oracle index types require declaring the operators that will trigger index scanning. In Oracle's conception, an operator links a user-defined function to an index type. For example, we defined an operator for range similarity query using the `Manhattan` distance function as follows:

```
CREATE OPERATOR Manhattan_dist BINDING (BLOB, BLOB)
  RETURN FLOAT USING Manhattan_distance;
```

The operator for the k-NN similarity query follows a similar approach. Having defined the operators and the implementation type, the new index types were created in a way similar as shown below:

```
CREATE INDEXTYPE Slim_Manhattan FOR
  Manhattan_dist(BLOB, BLOB), Manhattan_kNN(BLOB, BLOB)
  USING index_im_type;
```

After having the index types created, the Oracle database becomes aware of the new index. Therefore, the users are ready to create indexes over the application data based on the new index types, as exemplified below:

```
CREATE INDEX new_index ON image_table(image_sign)
  INDEX TYPE IS Slim_Manhattan PARAMETERS ('8192');
```

The single parameter required in our current implementation is the desired index page size (`8192`). Creating an index triggers the process illustrated in Figure 2. The query processor calls function

`ODCIIndexCreate` passing the new index information, such as the table and attribute to be indexed (Step 1). Function `ODCIIndexCreate` requests Arboretum to create an empty Slim-tree (Step 2). Thereafter, it queries the data (signature and RowId) stored in the table (Step 3). Using the signature LobLocators returned, the function fetches the BLOB data (Step 4) and inserts the signatures with the respective RowIds in the Slim-tree (Step 5). Finally, the function returns the control to the query processor (Step 6).

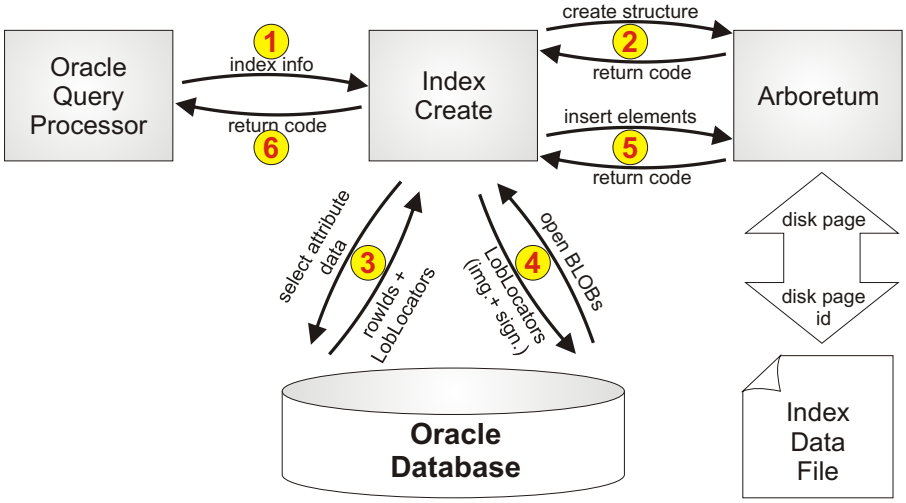

Fig. 2. Steps executed by function `ODCIIndexCreate`.

The created indexes are dynamically updated when insertions, deletions or updates occur on the underlying tables. The functions implementing such operations on the index (`ODCIIndexInsert`, `ODCIIndexDelete` and `ODCIIndexUpdate`) execute the steps 4-6 of the create index process, using the LobLocator and the RowId of the updated row provided by the query processor as parameters. When an index is dropped, function `ODCIIndexDrop` is called to remove the index.

The next subsection shows how the proposed indexes are used to execute similarity queries.

3.3 Execution of Indexed Range and k-NN Queries

To take advantage of the new indexes in a query, the associated operators need to be employed. For instance, an indexed range query must be written as in the following example:

```
SELECT * FROM image_table
WHERE Manhattan_dist(image_sign, center_sign) <= 0.5;
```

where `center_sign` is a BLOB containing the query center signature, the relational operator `<=` indicates that a range query has been requested, and the value `0.5` is the range radius. The index is used only if it is available and the first argument for the operator is the attribute of the table referred in the `FROM` clause. In other words, the index would not be used if the query was written using `Manhattan_dist(center_sign, image_sign)`. If any of these conditions is not satisfied, the query processor performs a sequential scan employing the underlying function (in this case, the `Manhattan_distance`).

In the same way, an indexed k-NN query is stated as follows:

```
SELECT * FROM image_table
WHERE Manhattan_knn(image_sign, center_sign) <= 10;
```

This command asks for the 10 nearest neighbors of the image whose signature is given as `center_sign`, considering the `Manhattan` distance.

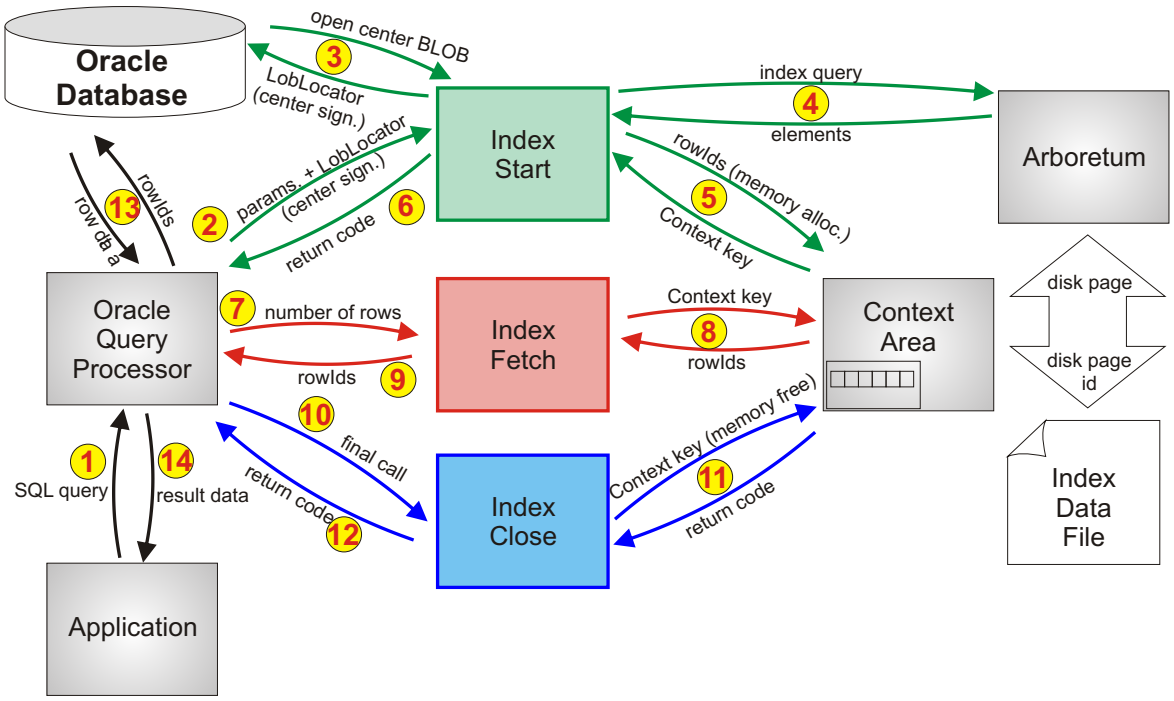

Fig. 3. Execution of an indexed similarity query in FMI-SiR$_O$.

The execution of an indexed similarity query in FMI-SiR$_O$ occurs as shown in Figure 3. The application submits a SQL query to the database manager (Step 1). The query processor identifies the similarity operator, which is a candidate for an indexed scan, and calls the function `ODCIIndexStart`, passing information about the operator, the index and the predicate used, the query center LobLocator and the search limits (Step 2). Function `ODCIIndexStart` loads the BLOB signature pointed by the query center LobLocator (Step 3) and executes the requested query over the index on Arboretum (Step 4). Then, the function saves the RowIds of the elements returned by the index in the current session context area, identifying the memory area by a unique key (Step 5). The context key is saved in attribute `scanctx` of the index type object to be used by the next functions called, and returns the control to the query processor (Step 6).

Thereafter, the query processor calls the function `ODCIIndexFetch`, providing the number n_{rows} of rows to be fetched (Step 7). Function `ODCIIndexFetch` gets the context key stored in the `scanctx` object attribute and retrieves the first n_{rows} RowIds stored in the context area (Step 8), returning them to the query processor (Step 9). If there are more tuples in the result, the query processor keeps calling function `ODCIIndexFetch` until the result set is completed.

When the fetch phase is finished, the query processor calls function `ODCIIndexClose` to end up the search (Step 10). Function `ODCIIndexClose` gets the context key and releases the memory allocated in the context area (Step 11), and then returns to the query processor (Step 12).

Finally, the query processor loads the row data pointed by the RowIds returned by the indexed scan, performs the remainder operations stated in the submitted SQL query and returns to the application.

As the module uses the Oracle's Extensible Interface, the query optimizer is aware of the indexed similarity search. Thus, it is possible to interleave its execution with other operations and choose the access paths properly. In an execution plan, the indexed similarity search is identified as a `DOMAIN INDEX` scan, as shown following for the previous k-NN example query:

```
|Id | Operation                   | Name                |
-----------------------------------------------------------
| 0 | SELECT STATEMENT            |                     |
| 1 |  TABLE ACCESS BY INDEX ROWID| ALOI                |
| 2 |   DOMAIN INDEX              | SLIM_MANHATTAN_ALOI |
```

3.4 Execution of Range Join Queries

FMI-SiR is also capable of answering similarity joins, performing range joins using both sequential and indexed access. A join operation combines tuples from two relations in a database, each one assuming the role either of the external or of the internal relation in the join operation. There are well-known basic algorithms in the literature to perform join operations. In this work we adapt two of them to perform similarity range join operations, the *nested loop join* and the *indexed loop join*.

The nested loop join is the naïve method to join two relations. For each tuple of the external relation, the nested loop algorithm scans the internal relation matching the tuples that satisfy the join condition. For two relations R and S, the similarity join algorithm performs $|R| * |S|$ distance calculations. Usually, the minor of the two relations is stated in the external loop to take advantage of the buffer management and achieve better performance. However, it is important to notice that the whole space of the internal relation is repeatedly explored at each step.

If one of the input relations possesses an index over the attribute and distance function employed in the range join, this index can be used to match tuples satisfying the join condition. The algorithm that employs this approach is the indexed loop join. It assigns the relation that has this index as the internal relation and executes over this index a range query centered in each tuple of the external relation. In each iteration, each tuple returned by the range query is concatenated to the respective tuple of the external relation, composing the result of the range join. The indexed loop strategy allows reducing the search space, exploring at each step only the region of the internal relation space centered in the current external tuple and delimited by the range join radius. The cost of the indexed loop join is $|R| * indexed\_Rq\_cost(S, \xi)$, where $|R|$ is the size of the external relation and $indexed\_Rq\_cost(S, \xi)$ is the average cost of indexed range queries over S with radius ξ. The cost of the indexed loop algorithm depends on the radius of the similarity join condition, but, in the great majority of the queries, this radius is small enough to outperform the nested loop method.

In FMI-SiR$_O$, the user can perform a similarity range join with the following clause, considering another table, `image_table2`, with structure similar to `image_table`:

```
SELECT out.id, inn.id
FROM image_table out, image_table2 inn
WHERE Manhattan_dist(inn.img_sign, out.img_sign) <= 0.5;
```

where `image_table` and `image_table2` are the input relations, `Manhattan_dist` is the operator employed, whose arguments are the signature attributes joining the two relations, and `0.5` is the join radius. The order of the parameters to the operator is important for efficient join execution. If there is an index regarding the distance function employed on the attribute provided as the first parameter (`inn.img_sign` in the example), it will be used to execute an indexed loop join algorithm. In this case, the external table would be `image_table` and the internal `image_table2`. If such index is not available, as well as the operator's underlying function is employed instead of the operator (e.g. `Manhattan_distance`), the nested loop algorithm is executed (See Section 3.3). Unfortunately, the Oracle's query processor is not capable to automatically choose the less costly option if the joining

attributes are indexed in both relations. Thus, it is the user's responsibility to pose the query that would lead to the most efficient execution.

4. EXPERIMENTS

We evaluated FMI-SiR$_O$ over several image datasets. In this section we present the results obtained using the *Amsterdam Library of Object Images* (ALOI)[3] [Geusebroek et al. 2005], which is a collection of color images from one thousand small objects, with 108 images of each object, varying illumination color, illumination angle and viewpoint, totalizing 108,000 images. Figure 4 shows a small sample of this dataset.

Fig. 4. A sample of the ALOI image collection.

We bulk-loaded the ALOI dataset images into the Oracle database, and for each image we generated four signatures: (i) the Oracle's native ORDImageSignature, (ii) a 256 bins Histogram for the Color descriptor, (iii) the first 256 Zernike moments for Shape, and (iv) the Haralick descriptors with 140 features for Texture. Thereafter, we performed several tests on the Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 64bit under Kubuntu GNU/Linux 8.10 64bit, running on a machine equipped with an Intel Core 2 Quad 2.83GHz processor, 4GB of RAM and a SATA2 HD of 750GB and 7,200RPM. The next section presents the results of these experiments.

4.1 Retrieval Quality

The quality of the retrieval was evaluated using Precision *versus* Recall graphs (P×R) [Baeza-Yates and Ribeiro-Neto 1999]. It was assumed that an image is similar to the query center if it is an image of the same object. The query recall is the fraction of the relevant elements that has been retrieved. The query precision is the fraction of the retrieved elements that are relevant. Thus, the closer to the top a P×R curve is, the better the result.

We evaluated several distance functions over each descriptor and selected the best for each one: L1 for both Color and Shape descriptors and Canberra for the Texture descriptor. Figure 5 presents the P×R graphs comparing the retrieval quality of our descriptors, implemented in Oracle, to the Oracle's corresponding one alone, considering individually Color, Shape and Texture (i.e., the evaluated descriptor was weighted with 1 and the remainders with 0). Analyzing the graphs of Figure 5, it is observed that the proposed approach clearly improves the precision of similarity queries in every case. Considering the Color descriptor, a precision gain of up to 50% was achieved in comparison with the precision obtained by the Oracle's descriptor. Regarding the Shape descriptor, FMI-SiR$_O$ reached up to 92% of precision gain considering a recall level of 5%. Our approach for Texture descriptor presented a notable gain in precision achieving a gain of up to 171% at the same recall level. It is important to emphasize that k-NN queries usually ask for small values of k. Thus, the small values of recall are the most relevant ones.

[3]Available at http://staff.science.uva.nl/∼aloi/

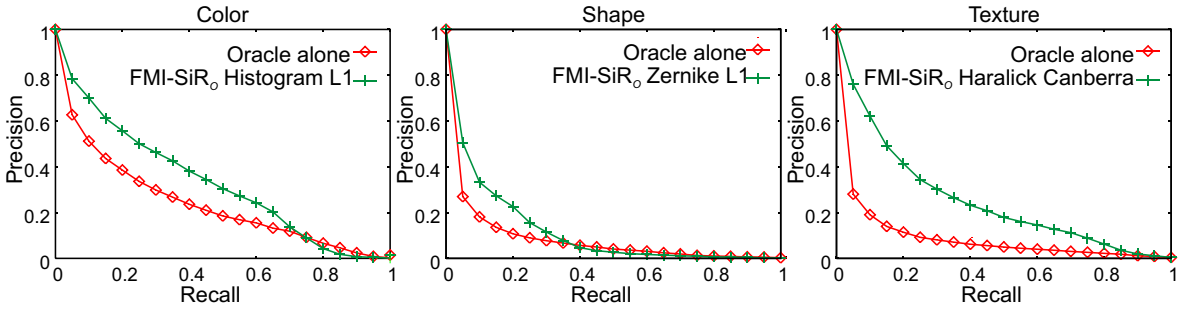

Fig. 5. P×R graphs comparing the retrieval quality of FMI-SiR$_O$ and Oracle alone.

4.2 Range and k-NN Query Performance Evaluation

We also performed analyses based on the time requirements to execute similarity queries. This subsection shows results regarding range and k-NN queries. It is important to emphasize that it is not possible to compare a FMI-SiR index using the Oracle's distance function, as the function implemented by Oracle is not a metric. All values presented herein were obtained by the average of 500 queries over the dataset.

Figure 6 shows the time required to perform k-NN queries, with k varying from 1 to 100. Notice that the time is shown in logarithmic scale. It can be observed that the FMI-SiR$_O$ sequential scan approach performs queries about twice slower than Oracle did, regardless of the value of k. It is due to the higher overhead to compute the distance functions, which are executed through external C++ functions embedded into a shared library, called repeatedly for each image. We verified that the cost of the arithmetic calculations over the features is almost irrelevant compared to the overall cost of the distance function call (less than 5% in fact), since the distances employed are cheap to compute (L1 and Canberra). The major computational cost is in the connection between Oracle and the external library and in the functions to handle BLOBs, being the latter responsible for more than 60% of the overall cost. However, as Oracle does not support indexed k-NN queries, our indexed k-NN search approach using the Color descriptor was able to perform 116 times faster than the Oracle sequential scan. Notable gains were also accomplished regarding the Shape and Texture descriptors, where FMI-SiR$_O$ was 71 and 175 times faster than Oracle alone, respectively.

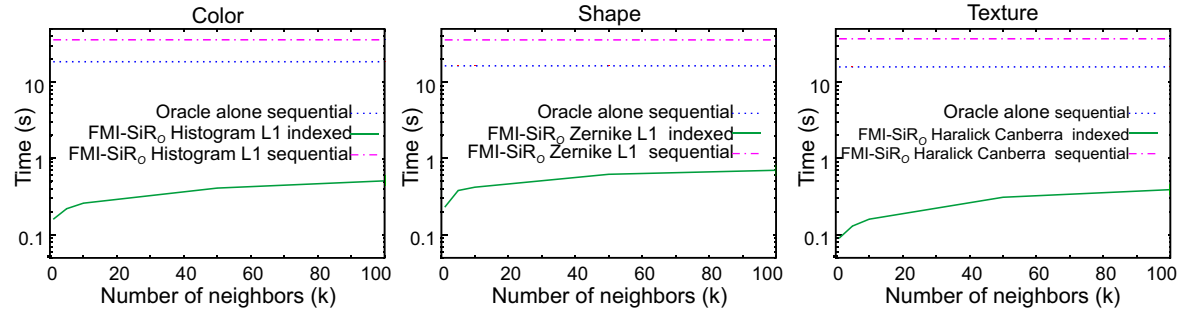

Fig. 6. Comparing k-NN queries execution times in FMI-SiR$_O$ and Oracle alone.

Regarding the range queries, we evaluated indexed and sequential accesses, both using only Oracle with interMedia and attaching FMI-SiR$_O$ over it. The radius for each range query was defined by the distance between the query center and its k-NN, in order to obtain k elements as a result. Figure 7 shows results for range queries. As occurred with the k-NN queries, the FMI-SiR$_O$ performs sequential scan about 2 times slower than the Oracle. However, our indexes allowed Oracle to be up to 2.7 times faster than using its internal index for the Color descriptor. Regarding the Shape descriptor, we again

achieved better results, as our index was up to 3.5 times faster than the Oracle's internal index. With regard to the Texture descriptor, both approaches tied for $k \leq 5$, then we lost. However, this is due to the Oracle's indexing approach, as the data obtained by its indexed access differs from those obtained by its sequential access.

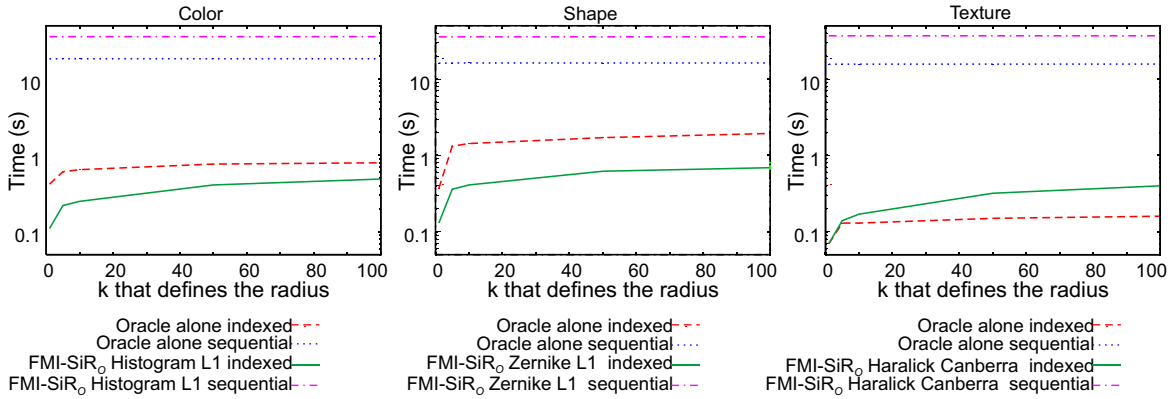

Fig. 7. Comparing range queries execution time in FMI-SiR$_O$ and Oracle alone, for both sequential and indexed access.

To illustrate the problem on Oracle's indexing approach, we evaluated the number of rows retrieved by each approach. The results are shown in Figure 8. The graphs show that neither the FMI-SiR$_O$ indexed scan nor the Oracle range sequential scan generate false negatives, returning 100% of the rows that satisfy the range predicate. However, the curves of Figure 8 reveal that the Oracle's index produces a huge amount of false negatives, considering any descriptor. For instance, using the Color descriptor, the Oracle index does not retrieve up to 47% of the answer set. Considering the Texture descriptor, which was the faster in the performance evaluation, it misses almost the entire dataset for large radii (missing up to 98% of the elements). Since the source code of Oracle interMedia is proprietary, we could not investigate why this occurs. However, it is important to highlight the fact that if an index generates false negatives, its answers become unreliable. Therefore, our indexed range scan must be compared to the Oracle sequential range scan, where Oracle enriched with the FMI-SiR$_O$ indexes performed up to 165, 210 and 123 times faster than the Oracle alone sequential range, with regard to Color, Texture and Shape respectively, with no misses.

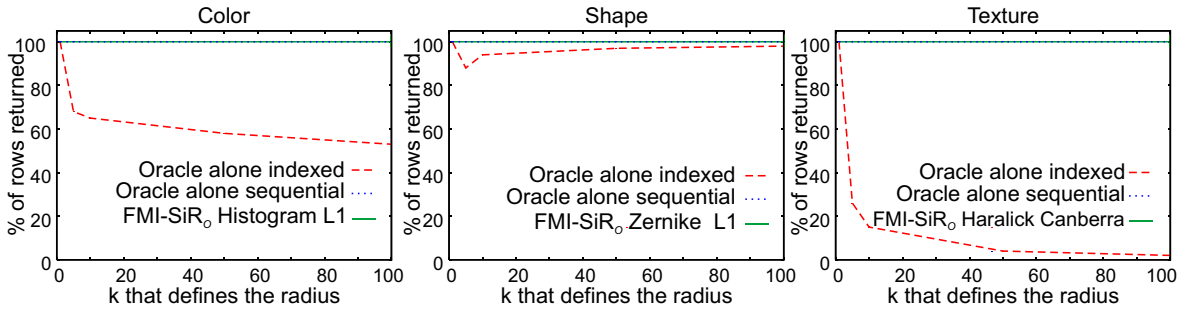

Fig. 8. Evaluation of false negatives generated by Oracle indexed retrieval compared to the sequential scan ground truth and to FMI-SiR$_O$ indexed scan.

Finally, we evaluated how much time is spent by the Oracle query processor during a search over a FMI-SiR$_O$ index and how much time is spent traversing the index. We compiled FMI-SiR$_O$ commenting the source code lines that perform the searches and just generate an empty result set. The queries over this "fake" indexed search took about 0.07 seconds in the average, which we can consider to be the overhead of calling our indexes. Notice that this overhead corresponds to from 16% to 44% and

from 10% to 18% of the elapsed time, in the average, respectively by the 10-NN and 100-NN queries executed over FMI-SiR$_O$ indexes in the experiments shown.

4.3 Range Join Performance Evaluation

This subsection shows results of tests regarding the time requirements and scalability of range join operations. In order to evaluate our approach regarding the time requirements we divided the ALOI dataset to compose the internal and external relations, according to each experiment's goal.

The first experiment aims at evaluating how the join cardinality affects the performance. We created a relation with 1K (1000) tuples and other with 10K (10000) tuples, which demands up to 10 million comparisons to perform a range join. Thereafter we performed range joins varying the join cardinality from 1K to 100K tuples. The comparisons of the time requirement were performed regarding Oracle-only sequential range join and the FMI-SiR$_O$ using the sequential (nested loop) and indexed range join (indexed loop). Despite Oracle alone supports indexed range join queries, it also produces false negatives. Thus, only the sequential approach is considered in the experiments.

In Figure 9 we can observe that the FMI-SiR$_O$ sequential range join approach again performs the queries about twice slower than Oracle alone, for the several join cardinality values. On the other hand, when analyzing the performance obtained by the FMI-SiR$_O$ indexed range join, a notable gain can be observed. The time is shown again in logarithmic scale. Our approach was up to 252 times faster than using Oracle alone for the Color descriptor. With regard to Shape and Texture descriptors, we again achieved outstanding results, as our approach was, respectively, up to 135 and 360 times faster than Oracle alone.

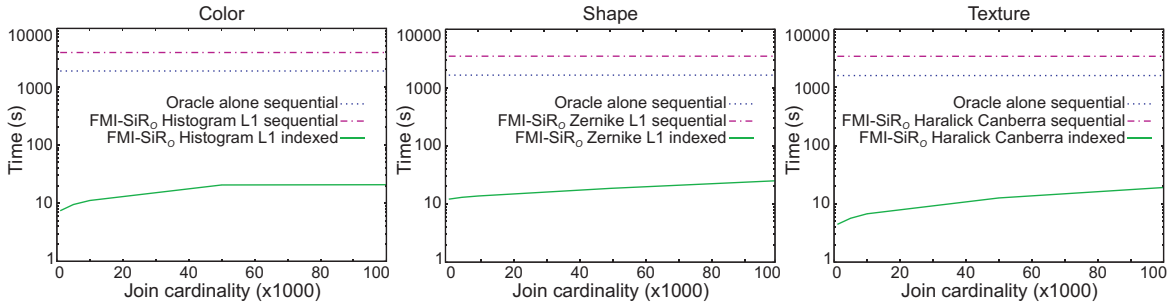

Fig. 9. Comparing range join queries execution times in sequential and indexed FMI-SiR$_O$ and Oracle alone with varying join cardinality.

With the purpose of evaluating the scalability of the FMI-SiR$_O$ indexed range join, we performed two experiments. In the first we execute the indexed range join varying the size of the internal relation from 1K to 108K tuples, maintaining the size of the external join relation in 1K tuples and the join cardinality in 10k tuples. As it can be seen in Figure 10, FMI-SiR$_O$ exhibits sub-linear behavior when the size of the internal relation grows, considering all descriptors, that makes the method adequate to execute indexed range join operations with very large internal relations. It is important to highlight that due to the prohibitive time execution for the sequential range join, it was not showed in the scalability experiments.

In the second scalability experiment, we set in the size of the internal join relation to 10K and varied the size of the external relation from 1K to 10k tuples. The join cardinality was also set in 10K tuples. Figure 11 shows that FMI-SiR$_O$ again presents a sub-linear behavior for growing external relation size.

The experiments show that the FMI-SiR$_O$ is adequate to perform range join operations in very large datasets, in any of its configurations. Moreover, comparing the results of Figures 10 and 11,

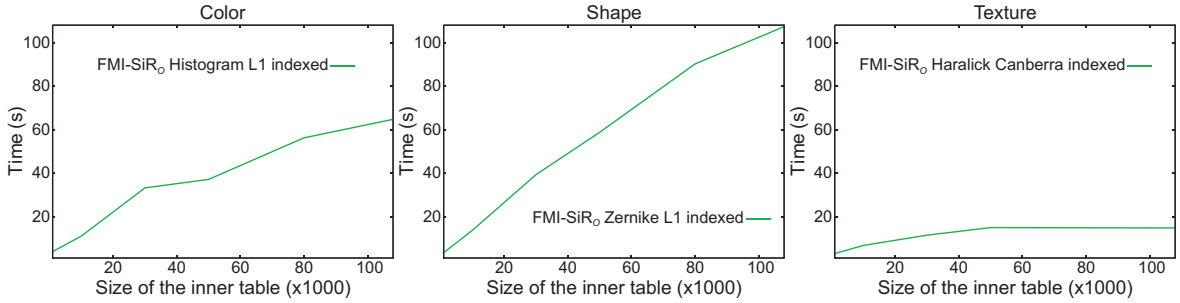

Fig. 10. Scalability of the range join query execution in FMI-SiR$_O$ for indexed access, varying the size of the internal relation.

we can notice that increasing the external relation size causes greater impact than the the internal relation size increasing, as FMI-SiR indexes scale better.

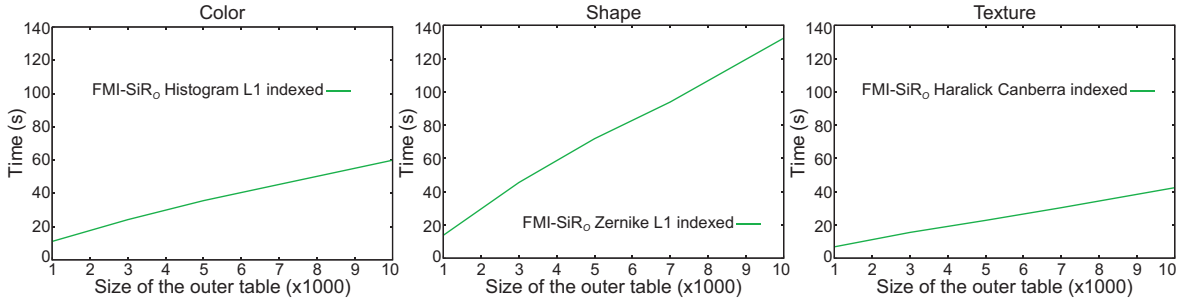

Fig. 11. Scalabity of FMI-SiR$_O$ join query execution for indexed access, varying the size of the external relation.

Considering the results achieved, we argue that our proposed approach not only accomplished high flexibility including new feature extraction methods and distance functions into the database core, but it is also well-suited to perform similarity queries improving the precision of them in a great extent while reducing their time requirements by two orders of magnitude or more. Moreover, our approach do not present false negatives.

5. CONCLUSIONS AND FUTURE WORK

Similarity searching is a research topic that has been investigated during several years. However, few tools exist making them available for industrial strength software production. Oracle released its interMedia module, aimed at providing image query-by-content resources for its DBMS. However, it is a proprietary tool and it does not allow users to provide their own definition of what is similarity, so users cannot adapt it for their own requirements. Moreover, the Oracle index generates many false negatives, turning the answer incompatible with that obtained by a sequential scan.

In this paper we presented an open source module for similarity search, called the FMI-SiR (user-defined **F**eatures, **M**etrics and **I**ndexes for **Si**milarity **R**etrieval over ***O***racle). Our approach, called the FMI-SiR$_O$, is implemented on top of the Oracle's Extensible Architecture Framework. It allows users to include new feature extraction methods and distance functions, to enhance domain specific data search as well as new index structures, without generating false negatives. Moreover, the developed module takes advantage of the Oracle query optimizer, making it possible to the user to mix regular and similarity operations, and the database engine to choose a good access path. We also presented experiments that shows that using Oracle with FMI-SIR$_O$ outperforms Oracle alone, both regarding the precision of similarity queries, with gains up to 171% in precision, and regarding performance, with indexed similarity selections being up to 175 times faster, and similarity range joins being up to

360 times faster, whereas presenting a sub-linear scaling regarding the size of the relations. Future work include to implement statistics gathering and cost models for similarity queries in the module, provide transaction support for FMI-SiR$_O$ indexes and develop additional similarity operators.

Acknowledgments

This work has been supported by CNPq, CAPES, FAPESP, STIC-AmSud and Microsoft Research.

REFERENCES

Baeza-Yates, R. A. and Ribeiro-Neto, B. A. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

Barioni, M. C. N., Razente, H. L., Traina, A. J. M., and Traina Jr., C. Seamlessly integrating similarity queries in SQL. *Software - Practice and Experience* 39 (4): 355–384, 2009.

Barioni, M. C. N., Razente, H. L., Traina Jr., C., and Traina, A. J. M. Querying complex objects by similarity in SQL. In *Proceedings of the Brazilian Symposium on Databases*. Uberlândia, Brazil, pp. 130–144, 2005.

Böhm, C. and Krebs, F. The k-nearest neighbor join: Turbo charging the kdd process. *Knowledge and Information Systems* 6 (6): 728–749, 2004.

Bugatti, P. H., Traina, A. J. M., and Traina Jr., C. Assessing the best integration between distance-function and image-feature to answer similarity queries. In *Proceedings of the ACM Symposium on Applied Computing*. Fortaleza, Brazil, pp. 1225–1230, 2008.

Burkhard, W. A. and Keller, R. M. Some approaches to best-match file searching. *Communications of the ACM* 16 (4): 230–236, 1973.

Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Data Bases*. Athens, Greece, pp. 426–435, 1997.

Flickner, M., Sawhney, H. S., Ashley, J., Huang, Q., Dom, B., Gorkani, M., Hafner, J., Lee, D., Petkovic, D., Steele, D., and Yanker, P. Query by image and video content: The QBIC system. *IEEE Computer* 28 (9): 23–32, 1995.

Geusebroek, J. M., Burghouts, G. J., and Smeulders, A. W. M. The Amsterdam library of object images. *International Journal of Computer Vision* 61 (1): 103–112, 2005.

Guliato, D., Melo, E. V., Rangayyan, R. M., and Soares, R. C. PostgreSQL-IE: An image-handling extension for PostgreSQL. *Journal of Digital Imaging* 22 (2): 149–165, 2008.

IBM Corporation. *Image, audio, and video extenders administration and programming guide*, 2003. DB2 Universal Database Version 8.

Informix Corporation. *Excalibur Image DataBlade Module User's Guide*. Informix Press, 1999.

ISO/IEC. *SQL - Part 2: Foundation*, 2003. Working Draft.

Kaster, D. S., Bugatti, P. H., Traina, A. J. M., and Jr., C. T. Incorporating metric access methods for similarity searching on Oracle database. In *Proceedings of the Brazilian Symposium on Databases*. Fortaleza, Brasil, pp. 196–210, 2009.

Long, F., Zhang, H., and Feng, D. D. Fundamentals of content-based image retrieval. In *Multimedia Information Retrieval and Management – Technological Fundamentals and Applications*. Springer, 2003.

Oracle Corporation. *Oracle interMedia User's Guide, 10g Release 2 (10.2)*. Oracle, 2005.

Skopal, T. and Lokoč, J. New dynamic construction techniques for M-tree. *Journal of Discrete Algorithms* 7 (1): 62–77, 2009.

Traina Jr., C., Traina, A. J. M., Faloutsos, C., and Seeger, B. Fast indexing and visualization of metric datasets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering* 14 (2): 244–260, 2002.

Vieira, M. R., Traina Jr., C., Traina, A. J. M., and Chino, F. J. T. DBM-tree: A dynamic metric access method sensitive to local density data. In *Proceedings of the Brazilian Symposium on Databases*. Brasília, Brazil, pp. 33–47, 2004.

Yianilos, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM/SIGACT-SIAM Symposium on Discrete Algorithms*. Austin, USA, pp. 311–321, 1993.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. *Similarity Search: The Metric Space Approach*. Advances in Database Systems, vol. 32. Springer, 2006.

Zhou, X., Wang, G., Zhou, X., and Yu, G. BM$^+$-tree: A hyperplane-based index method for high-dimensional metric spaces. In *Proceedings of the International Conference on Database Systems for Advanced Applications*. Beijing, China, pp. 398–409, 2005.