SQL-Based Compound Object Comparators: A Case Study of Images Stored in ICE

Dominik Ślęzak[1,2] and Łukasz Sosnowski[3,4]

[1] Institute of Mathematics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
[2] Infobright Inc.
Krzywickiego 34 lok. 219, 02-078 Warsaw, Poland
[3] Systems Research Institute, Polish Academy of Sciences
Newelska 6, 01-447 Warsaw, Poland
[4] Dituel Sp. z o.o.
Wąwolnicka 4 lok. 22, 04-023 Warsaw, Poland
slezak@infobright.com, l.sosnowski@dituel.pl

Abstract. We introduce the framework for storing and comparing compound objects. The implemented system is based on the RDBMS model, which – unlike other approaches in this area – enables to access the most detailed data about considered objects. It also contains ROLAP cubes designed for specific object classes and appropriately abstracted modules that compute object similarities, referred as comparators. In this paper, we focus on the case study related to images. We show specific examples of fuzzy logic comparators, together with their corresponding SQL statements executed at the level of pixels. We examine several open source database engines by means of their capabilities of storing and querying large amounts of such represented image data. We conclude that the performance of some of them is comparable to standard techniques of image storage and processing, with far better flexibility in defining new similarity criteria and analyzing larger image collections.

Keywords: Compound Objects, Similarity, Comparators, Image Analysis, Fuzzy Logic, RDBMS Engines, Infobright Community Edition (ICE).

1 Introduction

There is a growing demand for systems that can retrieve compound objects based on their mutual similarities, membership to certain groups, or satisfaction of some criteria. Such systems need to identify objects quickly and accurately, based on their comparison against some patterns or exclusion according to some forbidden features. In some applications, such systems work mainly with various types of objects' indexes and metadata. In other applications, they may also involve the objects' storage, which raises additional challenges but, on the other hand, opens new possibilities for the retrieval process improvements.

In this paper, we outline a framework that is able to retrieve objects basing on similarities, including their classification and interpretation. Similarities can be defined by a number of criteria and measures, varying with respect to the object classes and application types [13,18]. However, from the system's architecture standpoint, their implementation can be kept within a universal structure, referred here as a comparator. Actually, comparators occur in the literature in various contexts, usually for the purposes of image analysis and processing [4,10]. In our approach, however, the comparator is an abstracted module responsible for comparing collections of input objects (not necessarily images) and reporting outputs in the form of, e.g., parameters of the most similar objects. The comparator may be thus regarded as analogous to a mathematical function, whose values can be applied at further stages of analysis of objects.

An object is an entity that we want to measure, describe, classify or compare with other objects. It may correspond to a physical phenomenon, situation, state, process, signal, etc. It may have some features that are useful in classification or similarity analysis. Our understanding of an object is close to the concept of entity in relational databases or an object in information retrieval [7,15]. In this paper, we refer to objects at a possibly abstracted level, although it is useful to distinguish some specific classes of objects, such as images, texts or sequences. Variety of possible object classes is not in contradiction with universality of the proposed system's architecture. It is important because analogous solutions usually focus on implementation of algorithms dedicated to some particular types of objects that are not so easily transferrable to other cases.

The systems aimed at compound object retrieval usually assume a sharp distinction between the layers of storage and analysis. This means that the analytical algorithms have direct access only to precalculated features, often stored within an RDBMS framework, while the objects themselves are encoded as BLOBs or stored in an independent repository [5,9]. Initial phases of processing and segmentation in the image retrieval systems may be coupled with computation of the values of a pre-defined set of attributes based on histogram computation, edge detection, shape recognition, texture analysis, etc. Such an approach is quite convenient and, actually, it satisfies our above-formulated universality assumptions. However, it does not provide opportunity to efficiently refine and extend the set of features while applying new algorithms, as there is no direct interface to quickly manipulate the detailed data.

We propose an alternative approach to representing compound objects in relational data model. Namely, we put decoded information about objects into ROLAP cubes. For example, we suggest basing the cubes for images on the data table, where each pixel of each image is represented as a separate row. This way, the whole system takes the form of an integrated data warehouse. It enables a convenient access to arbitrary fragments of objects or collections of objects, as well as their analysis using standard database operations. Furthermore, it guarantees easiness of completing or modifying object descriptions in an arbitrary moment, not only at the stage of supplying objects to the system. Surely, the cubes and the underlying data need to be designed separately for different object classes. On the other hand, the usage of ROLAP operations by other system's components can be similar for all types of compound objects.

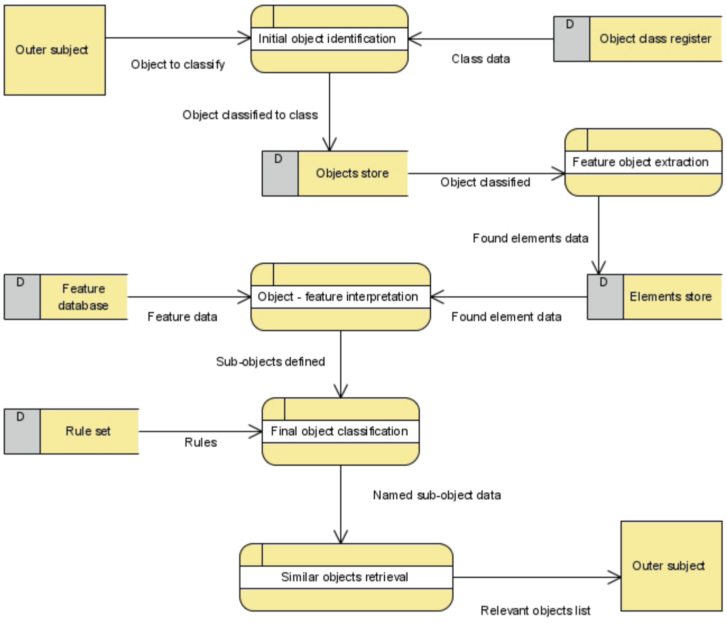

Fig. 1. DFD for compound object comparison

The remainder of the paper is organized as follows: In Section 2, we describe the proposed architecture with a special emphasis on the comparator aspects. In Section 3, we discuss the database framework aspects – compound object representation, the choice of appropriate technology and the underlying data schemas. In Section 4, we introduce a more specific case study of image comparator based on histogram analysis and fuzzy logic. In Section 5, we run some performance tests and analyze their results. In Section 6, we conclude the paper.

The presented approach is a continuation of our research in [19,20], extended here by a more complete study of the RDBMS layer. Thanks to pixel-based image representation, histogram computations and image comparisons, such as those in Sections 3 and 4, can be conducted using standard SQL syntax. Certainly, one might look at the idea of representing compound objects in such a detailed way as unrealistic, given the expected size of, e.g., large image databases. However, the results reported in Section 5 prove that modern analytic database engines provide fully satisfactory data compression and query performance.

2 Algorithmic Outline

In order to clearly present the proposed framework for compound object comparisons at a possibly universal level, we carefully follow the structural design and modeling standards [2,14]. Figure 1 presents a general Data Flow Diagram (DFD) for our solution. Table 1 provides more detailed information.

Table 1. Detailed description of components illustrated in Figure 1

Component name	Description
Outer subject	A user or a system that initializes and triggers identification/classification of an object by its comparison to the existing reference stores
Object class register	Detailed information about the types of object classes
Initial object identification	Recognizing the class of an object (e.g.: image, video, sound, text) in order to choose an appropriate set of comparison features and rules
Object store	Temporary storage of the investigated object
Feature object extraction	Using feature extraction techniques available for a given object class (e.g.: for images, it includes the edge detection, the histogram extraction, etc.)
Elements store	Temporary storage of the extracted elements (elements do not have a status of features yet)
Feature database	Types of features – specialized functions used to measure their values (e.g.: red color histogram)
Object-feature interpretation	Interpreting membership of the investigated object to the sets of elements with given major features (it can be conducted by using, e.g., fuzzy rule sets)
Rule set	Rules defining final classification of objects
Final object classification	Analyzing similarities between the identified elements of the investigated object and features in database; it enables to reject fake features and better interpret the remaining ones; final classification is designed to be conducted by an ensemble of comparators
Similar objects retrieval	Finding objects that are most similar to the investigated one, basing on its classification

Figure 2 presents the main stages of comparator's work. Comparing to Figure 1, this is a more atomic level of the proposed solution. The algorithm verifies similarity of an investigated object to the reference objects, including their elements too. There is also a mechanism of similarity exception handling. For each reference object b, we register its so called forbidden features. If the investigated object a turns out to have one of such features, then b cannot be reported as similar to a. Identification of forbidden features is based on the fuzzy classifier [6,12]. This way, we can take an advantage of domain knowledge and we obtain a convenient framework for the parameter tuning.

If b is not forbidden with respect to a, we compute the degree of similarity of a to b. As in [19,20], we use fuzzy logic apparatus also at this stage. We compute fuzzy membership of (a, b) to the similarity relation defined on compound objects. Definition of membership can be adjusted to reflect a general similarity within a given object class or, e.g., similarity of some specific aspects of objects. Membership can be represented as a function $\mu : R \times R \rightarrow [0, 1]$, where 0 and 1 mean total dissimilarity and similarity, respectively.

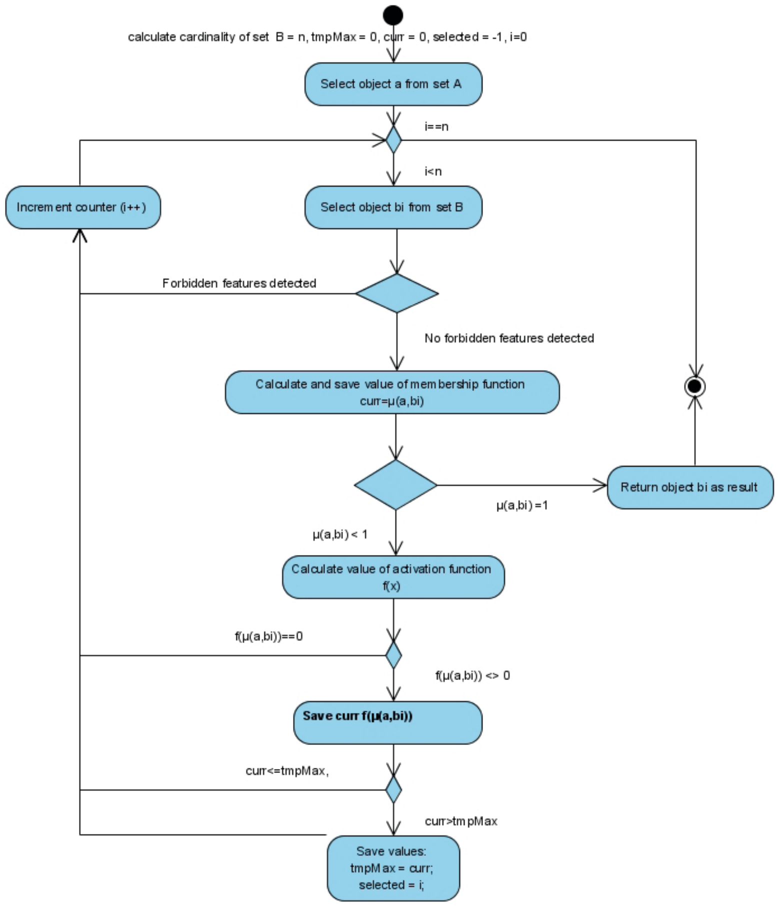
calculate cardinality of set B = n, tmpMax = 0, curr = 0, selected = -1, i=0
Select object a from set A
i==n
i<n
Increment counter (i++)
Select object bi from set B
Forbidden features detected
No forbidden features detected
Calculate and save value of membership function curr=μ(a,bi)
Return object bi as result
μ(a,bi) =1
μ(a,bi) < 1
Calculate value of activation function f(x)
f(μ(a,bi))==0
f(μ(a,bi)) <> 0
Save curr f(μ(a,bi))
curr<=tmpMax,
curr>tmpMax
Save values: tmpMax = curr; selected = i;

Fig. 2. Activity diagram of comparator

The degree of similarity can be further treated as an input to an activation function $f : [0, 1] \to \{0, 1\}$, which assigns 1 to values greater than a threshold $p \in [0, 1]$ and 0 otherwise. One may adjust p according to the expert knowledge or, e.g., as a result of evolutionary optimization process. Remember that various ways of computing function μ are just approximations of an actual notion of similarity between compound objects. Thus, one may search for p as the lowest possible threshold yielding results expected by the users.

In the last phase, we check whether the value of μ is higher than those computed so far. If it is, we memorize the corresponding reference object. Finally, we can get: a) no output (because of forbidden features or not exceeding threshold p), b) exactly one output, c) multiple outputs (if there are multiple objects b with the maximum value of $\mu(a, b)$). If we are interested only in a single output, we can stop after finding the first b satisfying $\mu(a, b) > p$.

3 RDBMS Framework

3.1 Compound Object Representation

The proposed design is based on defining different classes of compound objects that are equipped with comparable functionality. Usually, objects are available in various specific formats that are useful for storage but inconvenient for more advanced processing, when direct access to decoded data is required. In our solution, we want to enable the users to extract information from objects on ongoing basis, in order to provide better continuity of gathered knowledge, including methods of its gathering themselves. Knowledge and rule bases related to object classes should be allowed for evolving along with the users' needs. Hence, we decided to represent and store compound objects in their fully decoded form, in a relational database. Let us emphasize that we do not use such data types as BLOB, IMAGE, BYTE, as they do not address the above-mentioned issues. Instead, we operate with data schemas at the semantically richer level of atomic components of the compound objects.

For example, for collections of images, we suggest constructing data tables with rows corresponding to pixels. Each pixel is assigned with its coordinates within an image, as well as with its image identifier. Certainly, this means that even for low-resolution images the corresponding data table will grow very fast. In order to address this potential issue, we need to carefully select an underlying relational database engine technology.

3.2 Infobright Community Edition (ICE)

ICE[1] is an open source RDBMS engine integrated with MySQL. It may be applied in data warehouse or analytical database scenarios [16,17]. From the user perspective, it provides standard means for relational data modeling and querying. Internally, it decomposes data both vertically and horizontally, onto so called data packs, each of them storing values of up to 2^{16} rows for one of attributes. Data packs are compressed separately from each other. During load, besides compression, the content of each data pack is automatically annotated with its basic statistics that are stored in so called database knowledge grid. The acquired statistics are used in various ways in query execution, with the main goal of minimizing and optimizing an access to data packs.

ICE provides high compression ratio (reported as 10:1 on average) and high speed of analytical SQL statements, for which the gathered statistics are especially useful. ICE can easily handle tens of terabytes of data on a single PC machine. Also, it does not require maintenance of any additional database indexes, on top of database knowledge grid which is relatively small (reported as 1% of the compressed data size on average) and generated transparently to the users. Thus, given the challenges outlined in Subsection 3.1, we regard ICE as potentially applicable as the RDBMS layer of our solution.

[1] `en.wikipedia.org/wiki/Infobright`

3.3 Data Layout

Our solution is split onto two major parts: 1) OLTP (transactional layer) and 2) ROLAP (cubes dedicated to store information about objects). The OLTP part contains objects' metadata, their basic features, class membership, etc. It simplifies object management but has no direct impact on analytical capabilities. The ROLAP part resembles a data warehouse model. Cubes are created for specific data within object classes. Cubes may have partially common dimensions. This leads towards constellation schemas [1,7], which are convenient for more advanced analytics. Furthermore, we assume that cubes can be built for decoded objects, their specific elements or fragments, as well as for various types of pre-aggregates or statistics. Our solution enables to create such cubes in an arbitrary moment, depending on the users' requirements.

3.4 Relationships

Let us focus on simple examples of the above-mentioned components. The OLTP layer is displayed in Figure 3, by means of a standard Entity-Relationship Diagram (ERD). Table 2 describes particular entities. The data is stored in the third normal form in order to assure easiness of extensions and modifications. In the ERD diagrams, we use typical notation for foreign keys (FK) in order to emphasize joins that we expect to occur in SQL statements. However, there are no constraints / indexes assumed to be maintained.

In the ROLAP layer, each object class has its own star schema. Let us concentrate on the example of images, as illustrated by Figure 4 and Table 3. One can see that pixels are stored as rows in the fact table. Dimensions refer to coordinates X and Y, as well as to object identifiers. Such representation enables to express a number of image processing operations in SQL. For other useful types of operations we design specific stored procedures.

Figure 5 and Table 4 correspond to image histograms [3,11]. Histogram-based comparators can be quite efficient, as reported in Section 5. Histogram cube provides information about histograms of objects or their fragments. It can be automatically appended for each image being loaded into the database or computed at once by using, e.g., the following:

```
SELECT FK_OBJECTS_ID, 1 AS CHANNEL, RED AS
BRIGHTNESS, 1 AS IMG_AREA, COUNT(*) AS VALUE FROM
FACT_IMAGES GROUP BY FK_OBJECTS_ID, RED UNION ALL
SELECT FK_OBJECTS_ID, 2 AS CHANNEL, GREEN AS
BRIGHTNESS, 1 AS IMG_AREA, COUNT(*) AS VALUE FROM
FACT_IMAGES GROUP BY FK_OBJECTS_ID, GREEN UNION ALL
SELECT FK_OBJECTS_ID, 3 AS CHANNEL, BLUE AS
BRIGHTNESS, 1 AS IMG_AREA, COUNT(*) AS VALUE FROM
FACT_IMAGES GROUP BY FK_OBJECTS_ID, BLUE UNION ALL
SELECT FK_OBJECTS_ID, 4 AS CHANNEL, ALPHA AS
BRIGHTNESS, 1 AS IMG_AREA, COUNT(*) AS VALUE FROM
FACT_IMAGES GROUP BY FK_OBJECTS_ID, ALPHA
```

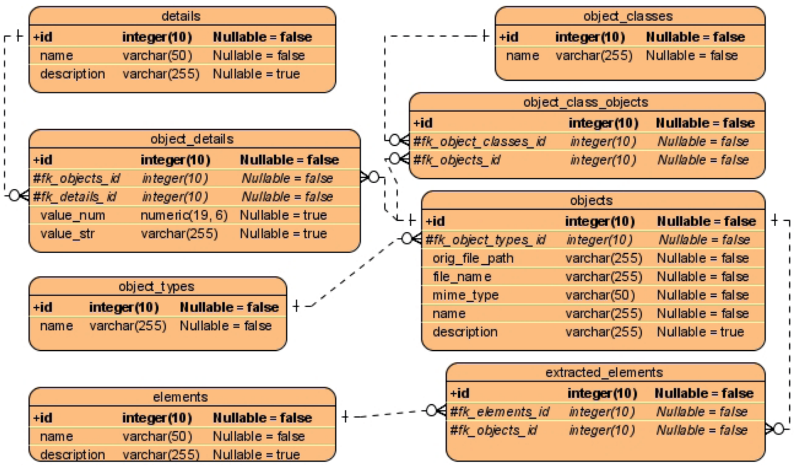
details
+id integer(10) Nullable = false
name varchar(50) Nullable = false
description varchar(255) Nullable = true
object_classes
+id integer(10) Nullable = false
name varchar(255) Nullable = false
object_class_objects
+id integer(10) Nullable = false
#fk_object_classes_id integer(10) Nullable = false
#fk_objects_id integer(10) Nullable = false
object_details
+id integer(10) Nullable = false
#fk_objects_id integer(10) Nullable = false
#fk_details_id integer(10) Nullable = false
value_num numeric(19, 6) Nullable = true
value_str varchar(255) Nullable = true
objects
+id integer(10) Nullable = false
#fk_object_types_id integer(10) Nullable = false
orig_file_path varchar(255) Nullable = false
file_name varchar(255) Nullable = false
mime_type varchar(50) Nullable = false
name varchar(255) Nullable = false
description varchar(255) Nullable = true
object_types
+id integer(10) Nullable = false
name varchar(255) Nullable = false
elements
+id integer(10) Nullable = false
name varchar(50) Nullable = false
description varchar(255) Nullable = true
extracted_elements
+id integer(10) Nullable = false
#fk_elements_id integer(10) Nullable = false
#fk_objects_id integer(10) Nullable = false

Fig. 3. ERD for OLTP layer

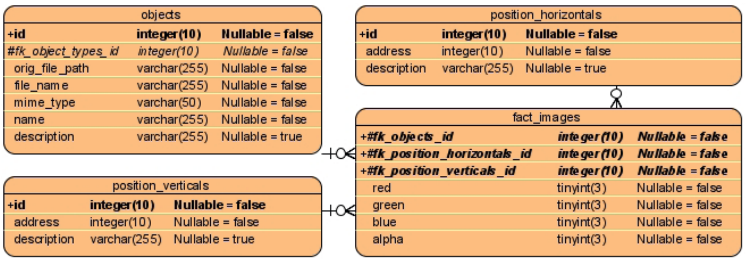
objects
+id integer(10) Nullable = false
#fk_object_types_id integer(10) Nullable = false
orig_file_path varchar(255) Nullable = false
file_name varchar(255) Nullable = false
mime_type varchar(50) Nullable = false
name varchar(255) Nullable = false
description varchar(255) Nullable = true
position_horizontals
+id integer(10) Nullable = false
address integer(10) Nullable = false
description varchar(255) Nullable = true
fact_images
+#fk_objects_id integer(10) Nullable = false
+#fk_position_horizontals_id integer(10) Nullable = false
+#fk_position_verticals_id integer(10) Nullable = false
red tinyint(3) Nullable = false
green tinyint(3) Nullable = false
blue tinyint(3) Nullable = false
alpha tinyint(3) Nullable = false
position_verticals
+id integer(10) Nullable = false
address integer(10) Nullable = false
description varchar(255) Nullable = true

Fig. 4. ERD for image data

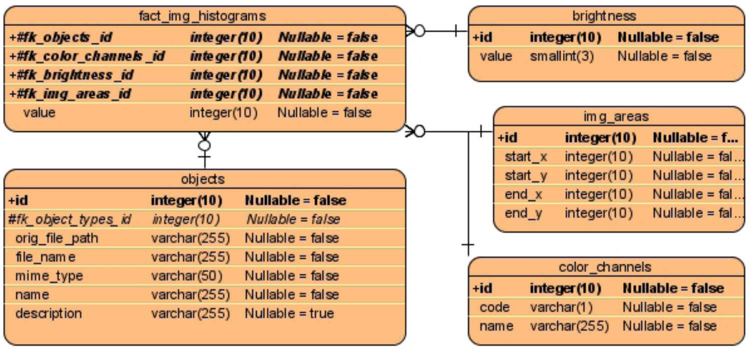
fact_img_histograms
+#fk_objects_id integer(10) Nullable = false
+#fk_color_channels_id integer(10) Nullable = false
+#fk_brightness_id integer(10) Nullable = false
+#fk_img_areas_id integer(10) Nullable = false
value integer(10) Nullable = false
brightness
+id integer(10) Nullable = false
value smallint(3) Nullable = false
img_areas
+id integer(10) Nullable = f...
start_x integer(10) Nullable = fal...
start_y integer(10) Nullable = fal...
end_x integer(10) Nullable = fal...
end_y integer(10) Nullable = fal...
objects
+id integer(10) Nullable = false
#fk_object_types_id integer(10) Nullable = false
orig_file_path varchar(255) Nullable = false
file_name varchar(255) Nullable = false
mime_type varchar(50) Nullable = false
name varchar(255) Nullable = false
description varchar(255) Nullable = true
color_channels
+id integer(10) Nullable = false
code varchar(1) Nullable = false
name varchar(255) Nullable = false

Fig. 5. ERD for image histograms

Table 2. Description of entities visible in Figure 3

Entity name	Description
object_classes	Related to the meaning or context, such as people, architecture, holiday photos etc.; independent from categories of types or formats, such as text, image etc.
elements	Dictionary of elements/features extracted from objects
object_class_objects	Assignment of objects to (possibly multiple) classes
objects	It includes information about objects' types; on the other hand, it serves as dimension for ROLAP cubes
object_types	Dictionary of object types
extracted_elements	Assignment of elements to objects
object_details	Features that describe objects' details
details	Dictionary of possible objects' details

Table 3. Description of entities visible in Figure 4

Entity name	Description
position_horizontals	Horizontal coordinates of pixels in their images
position_verticals	Vertical coordinates of pixels in their images
objects	Refers to "objects" in Table 2
fact_images	Fact table – rows correspond to images' pixels

Table 4. Description of entities visible in Figure 5

Entity name	Description
color_channels	Channel dimension (red R, green G, blue B, alpha A)
img_areas	Dimension defining areas that the histogram refers to (the whole image or, e.g., an object visible at an image)
brightness	Dimension defining pixels' brightness levels
objects	Refers to "objects" in Table 2
fact_img_histograms	Fact table for histogram-based ROLAP cube

The histogram cube contains up to 256×4 rows per image (256 brightness levels, 4 channels). Histograms can be further quantized by rounding their values to multiples of n. For instance, for $n = 10$, quantization can look as follows:

```
SELECT FIH.FK_OBJECTS_ID, CC.CODE, B.VALUE - (B.VALUE mod 10),
SUM(FIH.VALUE) FROM FACT_IMG_HISTOGRAMS FIH INNER JOIN BRIGHTNESS
B ON FIH.FK_BRIGHTNESS_ID = B.ID INNER JOIN COLOR_CHANNELS CC
ON FIH.FK_COLOR_CHANNELS_ID = CC.ID WHERE FIH.FK_IMG_AREAS_ID = 1
GROUP BY FIH.FK_OBJECTS_ID, CC.CODE, B.VALUE - (B.VALUE mod 10)
```

Quantization simplifies further steps of image comparison. If the histogram cube is already in place, the above SQL runs in milliseconds. On the other hand, we noticed that $n = 10$ does not lead to a significant decrease of accuracy.

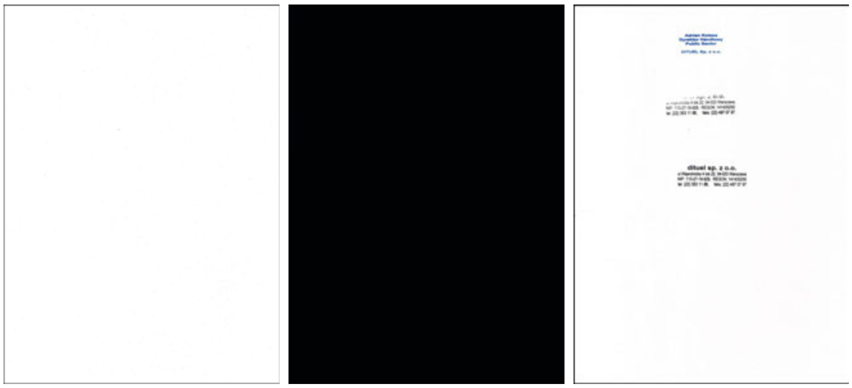

Fig. 6. Images #1, #2, #3

4 Case Study

Consider images #1,#2,#3 presented in Figure 6. They have resolution 1024×1408, which means 1441792 rows per image in the database. Table 5 shows quantization results for $n = 10$. Each image yields at most 26 rows. Images #1 and #2 are almost white and black, respectively. This explains why the first row for #1 and the last row for #2 have both very large values.

Consider comparators K_R, K_G and K_B corresponding to the color channels Red, Green and Blue, respectively.[2] Each K computes similarity of an investigated image to a reference image. Let us use the following function:

$$\mu(a,b) = 1 - \frac{\sum_{j=1}^{26} |a[j] - b[j]|}{2} \tag{1}$$

where $a[j]$ ($b[j]$) denotes the j-th index of the quantized and normalized histogram vector for the investigated (reference) image. The values of $\mu(a,b)$ are then treated as degrees of membership in fuzzy rules applied to final interpretation of similarity. We use conjunctions of memberships related to RGB.

Assume that image #3 is the investigated object. Let us compare it with the reference images #1 and #2 by following the approach outlined in Section 2. For simplicity, assume that there are no forbidden features. Set up $p = 0.999999$. By putting information from Table 5 into equation (1), for K_R, we obtain:

$$\mu(\#3,\#1) = 0.980630 \quad \mu(\#3,\#2) = 0.000006$$

Although image #3 is far more similar to #1 than to #2, the output of the algorithm displayed in Figure 2 is empty, because the result of K_R still needs to be combined within a conjunction with those of K_G and K_B, with no chance to exceed the threshold of 0.999999. One more time, we refer to [19,20] for details on how to adjust parameters of the proposed comparator methodology.

[2] Alpha can be omitted for jpg files. We plan to analyze other formats in the future.

Table 5. Quantized histograms for images #1 (top left), #2 (middle left) and #3

#1	R	G	B
40	1	1	1
70	1	1	1
160	3	3	3
190	0	0	2
200	2	2	5
210	5	5	9
220	14	16	14
230	25	26	34
240	332	331	387
250	1441409	1441407	1441336

#2	R	G	B
0	1441264	1441263	1441260
10	524	524	520
20	2	3	10
40	1	1	1
50	1	1	1

#3	R	G	B
10	4	1	1
20	34	1	1
30	337	39	24
40	1006	251	166
50	1497	851	610
60	1800	1456	1208
70	1835	1804	1680
80	1292	1727	1344
90	934	1577	864
100	767	1114	655
110	702	829	625
120	699	746	567
130	705	753	639
140	606	627	714
150	672	650	865
160	664	643	1050
170	678	694	1148
180	765	775	1108
190	834	837	1018
200	800	777	911
210	972	948	925
220	1180	1097	1128
230	1772	1392	1394
240	7756	6060	7805
250	1413481	1416143	1415342

5 Performance Tests

We report the results obtained on a standard laptop, Intel Core Duo T9600, 8GB RAM, 500GB 5400 RPM, Windows 7 64 bit.

The first part of our tests refers to images in Figure 6. Table 6 shows the size of their jpg files and the size of their corresponding sets of pixels stored in ICE. Compression ratios in ICE are worse than in case of dedicated jpg format but the difference is less than one might expect, especially for compression algorithms that are by definition lossless and lossy, respectively.

The speed of decoding a jpg file into a csv file and then loading it into ICE is on average 3,702 milliseconds. It may be improved in the future by avoiding creation of intermediate files. We repeated the tests twice: 1) for initially empty database and 2) for database containing initially around 300,000,000 rows, which corresponds to around 2,000 images of the considered resolution. The ICE load speed was approximately the same in both scenarios.

The last two columns in Table 6 reflect the speed of creating and quantizing histograms in Java [11,19] and ICE (using SQL similar to those in Subsection 3.4 but referring to each image separately). ICE speed is reported for 300,000,000^{+}-row data. In case of Java, it means finding a required jpg among 2,000 of other files and opening it to extract a quantized histogram.

Table 6. Image sizes (3 cases) and histogram creation + quantization speed (2 cases)

#	Size of jpg	Size of csv	Size in ICE	Java Speed	ICE Speed
1	13 KB	38,196 KB	91 KB	201 ms	1,035 ms
2	10 KB	29,750 KB	34 KB	214 ms	850 ms
3	78 KB	38,174 KB	770 KB	200 ms	1,170 ms

Table 7. Comparison of RDBMS engines for 3 × 1441792-row data. Results averaged over images #1,#2,#3. Last column refers to histogram creation + quantization speed.

	Load Speed	Database Size	Execution Speed
ICE 3.3	3,702 ms	298 KB	910 ms
MySQL 5.0	2,803 ms	40,832 KB	7,732 ms
PostgreSQL 8.4	17,803 ms	83,616 KB	20,755 ms

Table 8. Finding 10 out of 100/500/1,000 images. – Example of the search criterion.

Amount of Images	Search Speed in Java	Search Speed in ICE
100	19,310 ms	101 ms
500	89,349 ms	127 ms
1,000	181,474 ms	142 ms

Table 7 shows why we recommend ICE. Here, we consider only three images, as we had problems with 300,000,000 rows in other engines. Recall that ICE does not need indexes, as they are replaced by much lighter statistics that are, actually, very helpful when executing the discussed queries. In case of MySQL[3] and PostgreSQL[4], for better performance, one might use indexes. However, they would cause further increase of size and maintenance effort.

Going back to Table 6, one may claim that a standard approach is still faster than the RDBMS-based methodology. However, it turns out quite opposite in case of typical search processes – the subject of the second part of our tests. Table 8 illustrates the performance of our overall solution when the task is to find 10 out of 100, 500, or 1000 images that match to the highest degree some pre-defined conditions. Precisely, we search for images with a large number of pixels with the brightness = 250 for the blue channel, that is:

```
SELECT FK_OBJECTS_ID FROM FACT_IMG_HISTOGRAMS WHERE
FK_COLOR_CHANNELS_ID = 3 AND FK_BRIGHTNESS_ID = 250
ORDER BY VALUE DESC LIMIT 10
```

Surely, it is not as complicated as queries that we may encounter for advanced comparators. However, it illustrates how to use SQL efficiently.

[3] `dev.mysql.com/doc/refman/5.0/en/index.html`

[4] `www.postgresql.org/docs/8.4/static/index.html`

6 Conclusions and Discussion

We proposed a novel approach to storing and analyzing compound objects, where the object processing and comparison algorithms are able to run over complete, atomic data in a relational database model. We discussed high-level ideas, as well as technological details such as, e.g., the choice of Infobright Community Edition (ICE) as the underlying RDBMS solution. Comparing to our previous research [19,20], we focused on overall performance, paying less attention to analytical accuracy. The main goal of this paper was to provide a data representation framework that is flexible enough to tune advanced algorithms working efficiently with large collections of objects.

The results presented in Section 5 may be insufficient to fully convince everyone that our approach is worth considering. For example, given Tables 6 and 8, one might take an advantage of both standard and RDBMS-based solution by keeping images in their specific format and, as a complement, storing their histogram information in a relational data schema. However, as discussed in the earlier sections, the major reason for storing data about compound objects in the decoded form is to achieve better flexibility in feature extraction and comparison strategies. The already-mentioned results are supposed to prove that our solution should not be disqualified because of unacceptable compression or execution of the most typical operations. On the other hand, it clearly leads towards functionality that is beyond other methods.

In the nearest future, we will investigate opportunities that our framework provides at the level of processing multiple compound objects. Consider an example of images. Practically all the existing technologies assume that images are processed separately. Surely, it enables to parallelize the most time-consuming index and feature extraction operations. However, all further stages of, e.g., image comparisons need to be based on the extracted information instead of complete data. Among applications that may suffer from such limitation, one may look at, e.g., video analysis or 3D MRI brain segmentation [3,8]. More generally, the data layout proposed in Section 3 enables to run arbitrary SQL statements over atomic data related to an arbitrary subset of objects.

The knowledge about compound objects may be also employed to improve the database engine efficiency and functionality. In this paper, we use compound object hierarchies explicitly, at the data schema level. Alternatively, such knowledge can be expressed internally, transparently to the users and modules communicating with a database via SQL. Both strategies should be taken into account depending on practical needs. They may lead to better domain-specific compression, domain-specific statistics and also new ways of understanding SQL (see e.g. [6,16] for further discussion and references).

Acknowledgment. The first author was partially supported by the grants N N516 368334 and N N516 077837 from the Ministry of Science and Higher Education of the Republic of Poland.

References

1. Agosta, L.: The Essential Guide to Data Warehousing. Prentice Hall PTR, Englewood Cliffs (2000)
2. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, 2nd edn. Addison-Wesley Professional, Reading (2005)
3. Bovik, A.C. (ed.): Handbook of Image and Video Processing, 2nd edn. Academic Press, London (2005)
4. Cantu-Paz, E., Cheung, S.S., Kamath, C.: Retrieval of Similar Objects in Simulation Data Using Machine Learning Techniques. In: Proc. of Image Processing: Algorithms and Systems III, SPIE, vol. 5298, pp. 251–258 (2004)
5. Datta, R., Joshi, D., Li, J., Wang, J.Z.: Image Retrieval: Ideas, Influences, and Trends of the New Age. ACM Comput. Surv. 40(2), 1–60 (2008)
6. Galindo, J. (ed.): Handbook of Research on Fuzzy Information Processing in Databases. Information Science Reference (2008)
7. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book, 2nd edn. Prentice Hall PTR, Englewood Cliffs (2008)
8. Khotanlou, H., Colliot, O., Atif, J., Bloch, I.: 3D brain tumor segmentation in MRI using fuzzy classification, symmetry analysis and spatially constrained deformable models. Fuzzy Sets Syst. 160(10), 1457–1473 (2009)
9. Lew, M.S., Sebe, N., Djeraba, C., Jain, R.: Content-based Multimedia Information Retrieval: State of the Art and Challenges. ACM Trans. Multimedia Comput. Commun. Appl. 2(1), 1–19 (2006)
10. Lorenz, A., Blüm, M., Ermert, H., Senge, T.: Comparison of Different Neuro-Fuzzy Classification Systems for the Detection of Prostate Cancer in Ultrasonic Images. In: Proc. of Ultrasonics Symp., pp. 1201–1204. IEEE, Los Alamitos (1997)
11. Lyon, D.A.: Image Processing in Java. Prentice Hall PTR, Englewood Cliffs (1999)
12. Melin, P., Kacprzyk, J., Pedrycz, W. (eds.): Bio-Inspired Hybrid Intelligent Systems for Image Analysis and Pattern Recognition. Springer, Heidelberg (2010)
13. Pękalska, E., Duin, R.P.W.: The Dissimilarity Representation for Pattern Recognition: Foundations and Applications. World Scientific, Singapore (2005)
14. Rajan, S.D.: Introduction to Structural Analysis & Design. Wiley, Chichester (2001)
15. Sarawagi, S.: Information Extraction. Foundations and Trends in Databases 1(3), 261–377 (2008)
16. Ślęzak, D.: Compound Analytics of Compound Data within RDBMS Framework – Infobright's Perspective. In: Proc. of FGIT. LNCS, vol. 6485, pp. 39–40. Springer, Heidelberg (2010)
17. Ślęzak, D., Eastwood, V.: Data Warehouse Technology by Infobright. In: Proc. of SIGMOD, pp. 841–845. ACM, New York (2009)
18. Smyth, B., Keane, M.T.: Adaptation-guided Retrieval: Questioning the Similarity Assumption in Reasoning. Artif. Intell. 102(2), 249–293 (1998)
19. Sosnowski, Ł.: Intelligent Data Adjustment using Fuzzy Logic in Data Processing Systems (in Polish). In: Hołubiec, J. (ed.) Systems Analysis in Finances and Management, vol. 11, pp. 214–218 (2009)
20. Sosnowski, Ł.: Constructing Systems for Compound Object Comparisons (in Polish). In: Hołubiec, J. (ed.) Systems Analysis in Finances and Management, vol. 12, pp. 144–162 (2010)