cIPT: Shift of Image Processing Technologies to column-oriented Databases

Tobias Vinçon[1,2], Ilia Petrov[2,3], and Christian Thies[3]

[1] Hewlett Packard Enterprise, Böblingen, Germany
firstname.sirname@hpe.com
[2] Data Management Lab, Reutlingen University, Germany
[3] Reutlingen University, Germany
firstname.sirname@reutlingen-university.de

Abstract. The amount of image data has been rising exponentially over the last decades due to numerous trends like social networks, smartphones, automotive, biology, medicine and robotics. Traditionally, file systems are used as storage. Although they are easy to use and can handle large data volumes, they are suboptimal for efficient sequential image processing due to the limitation of data organisation on single images. Database systems and especially column-stores support more stuctured storage and access methods on the raw data level for entiere series.

In this paper we propose definitions of various layouts for an efficient storage of raw image data and metadata in a column store. These schemes are designed to improve the runtime behaviour of image processing operations. We present a tool called column-store Image Processing Toolbox (cIPT) allowing to easily combine the data layouts and operations for different image processing scenarios.

The experimental evaluation of a classification task on a real world image dataset indicates a performance increase of up to 15x on a column store compared to a traditional row-store (PostgreSQL) while the space consumption is reduced 7x. With these results cIPT provides the basis for a future mature database feature.

1 Introduction

Industries like automotive, biology, medicine and robotics produce more media data than ever nowadays. Trends in mobile devices and social media platforms additionally increase the amount of image data dramatically. Furthermore, the need to process this image data has risen up to real time applications in the last years, driven mainly by big data analyses [3].

Especially, the uniform analysis of image sequences is very wide-spread across emerging image processing applications for instance content based image retrieval, pedestrian tracking in intelligent cars or cell recognition in series of micrographs. Most traditional image processing tools use file systems, which are suboptimal persistence layers because algorithms cannot operate sequentially on specific data sections over all images without costly loading of every single image entirely into memory. DBMSs can make this data access more transparent with respect to local structures and thus offer the possibility to execute operations more efficient and closer to the data. However, traditional row-stores cannot efficiently handle the high volume data streams resulting from image sequences. Modern column-stores are well-suited for the needs of big data analysis [1].

This paper proposes the column-store Image Processing Toolbox (cIPT) by providing the ability to process image sequences on a column-oriented DBMS. cIPT allows to use different data layouts for the image data extracted from an input sequence. These data layouts leverage the characteristics of a column-store with a set of basic image processing operations. This approach is illustrated by implementing an image analysis pipeline for image classification which is the underlying principle of the applications mentioned above.

Data comes from the Cape2Cape project [4] (a cooperation between the German car manufacturer Volkswagen and Hewlett Packard) where a customized production car drove from the north to the south cape in under 9 days. The vehicle was equipped with a front camera configured to take high definition images every 10 seconds. The goal is to analyse the images in near-time, to extract information about the weather condition at the respective location and to correlate them with global weather forecasts. The result is: high volume image data, the need for optimal persistence and acceleration of image processing operations. The images have to be classified with respect to typical weather conditions.

The required image analysis pipeline is adapted to characteristics of the column-oriented database where the first version of cIPT includes: an image load utility, data layout converters, neighbourhood access functionality, extraction and comparison of histogram data as well as support for different distance metrics.

In this sense, cIPT provides a basic toolkit functionality for building image processing applications based on columnwise data access. Further contributions of this paper are the implementation of cIPT within a commercial column-store. The performance is evaluated on the real Cape2Cape image data set, which amounts to 183GB and comprises to approx. 70000 images. The experimental evaluation compares cIPT on a column-store against a traditional row-store (PostgreSQL).

The rest of this paper is organized as follows. The next Section presents the related work. The architecture of cIPT and the different data layouts as well as the definition of the operations is described in Section 3. The experimental evaluation is presented in detail in Section 4. We summarise our results and conclusions in Section 5.

2 Related Work

Since images are not only analysed individually but are rather considered as a source of observation data, their processing finds increasing application in many industrial areas [6][2].

The shift of the persistence layer of media storage applications from a conventional file storage to a relational database began in the early 90s with IBM's

QBIC project [8]. It provides an automatic feature extraction of loaded images and a similarity search on their basis. The runtime of utilized image processing algorithms is improved significantly by placing them in the DBMS. By storing only the extracted features like histograms, QBIC applications are able to search images within large collections but cannot efficiently perform repeating feature calculations with varying settings.

Recent research in the area of DBMS proves that column-oriented databases outperform traditional row-stores when handling data-intensive workloads as demonstrated by C-Store (nowadays Vertica)[10][7]. Efficiently utilised techniques like data compression, encoding, late materialisation, main-memory processing and support for parallelisation characterise the architecture of column-stores. Modern column-stores are able to handle mixed loads and update-intensive operations while performing complex analytical queries by applying various optimizations (e.g. the distinction between read- and write-optimised stores in Vertica or delta stores in other stores, etc.). MonetDB is a widely used open-source column-store. [5] illustrates that processing approximately 4 TB of image data within the Sloan Sky Server project, MonetDB has lower response times than a commercial row-store (MS SQL Server) for almost all queries.

Due to their characteristics, column-stores are suitable to process high volume image sequence data. Under cIPT we purpose an approach for a basic toolset of data layouts and operations as basis for image processing and detection algorithms. cIPT is optimised for column-oriented DBMSs and it has the potential for a major database feature in the future.

3 Architecture of cIPT

The column-store Image Processing Toolbox (cIPT) is a generic set of data layouts and algorithms for the purpose of image processing on column-oriented databases. Although existing functionality focuses on the comparison of images by calculating similarities in their colour values, further more general research questions on image processing on column stores can be answered. Moreover, cIPT has a modular and extensible design.

Features e.g. colour histograms, are derived from raw pixel data for instance for similarity search [2]. This toolbox currently uses RGB colour values and their average greyscale. The distribution of these values over all image's pixels is represented by histograms. If the application needs to extract a specific part of an image, cIPT will be able to filter the raw data very fast.

Usually, image processing algorithms behave in a common manner, based on a general workflow. The following enumeration sketches its steps, data interaction and its utilisation within the cIPT.

1. **Load** The load operation of cIPT extracts the RGB colour values of each pixel using an integrated C++ image processing API. If necessary, further information like the greyscale is calculated on the fly. The entire data is represented as vectors in the Tables *(rgb)*, *(rgbgrey)* or *(grey)*. This complete process is covered in a special module called User Defined Load (UDL).

2. **Convert** ***(optional)*** The conversion of image data from one relation to another is an optional operation that requires further calculations in special cases. Several conversion operations are designed within cIPT.
3. **Filter** ***(optional)*** Filtering is yet another optional operation within cIPT. It takes an image data table as input and extracts image data within certain shapes (e.g. rectangles, circles, etc.). The result has the input format, allowing pipelining.
4. **Calculate Histogram** The calculation of histograms is influenced by several execution parameters. Using minimum, maximum and the number of bins the detail degree is adjustable. These histograms are persisted and reused in further processing steps.
5. **Compare** The similarity of histograms can be calculated in several ways. For instance, the distance metric can weight big divergences much more than small ones or conversely. There are several similarity metrics available in cIPT (manhatten, euclidean, etc.).

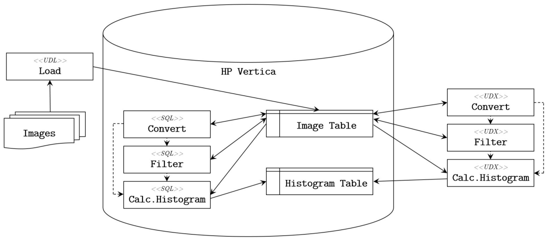

Fig. 1: General overview of relations and operations within the cIPT

Possible interactions within cIPT are sketched in Figure 1. cIPT includes **Image Tables** as raw data source and **Histogram Tables** as storage for calculated features. Both types of relations are discussed in detail in Section 3.1.

3.1 Data Schema Layout

The cIPT schema comprises two types of relations. Firstly, the image table stores images raw data, more precisely the colour representations *rgb*, *rgbgrey* and *grey*. Secondly, the histogram table stores the calculated histogram data including their properties. Tuples of both relations can be arranged in a horizontal, vertical or in partitioned manner causing performance differences. Although a partitioned layout might be efficient in some cases where queries only access data of one partitioned table as shown in [9], the cIPT usually accesses either the complete data of an image or an undefined portion of it. Therefore, partitioned layouts do not suit cIPT operations.

In addition to the payload, each table reserves space for an unique identifier and further metadata as shown exemplarily for the vertical *rgbgrey* layout in Table

1[4]. Unfortunately, the commercial column store we used for the performance evaluation is limited to a maximum of 1600 columns which is the reason an unpartitioned-horizontal layout is partially possible. If implemented, one image is represented by a single tuple comprising columns for each pixel. Because of the limitation, only images with resolution of smaller than 40x60 pixels can be stored which is obviously not suited for modern applications. However, histogram data can be arranged both vertically and horizontally. The size of cIPT's histograms depends primarily on the colour depth. 256 Containers are sufficient to store images with 8 bit colour depth. In addition, characteristics like the colour channel, the number of bins, minimum and maximum colour values are stored. An example of the vertical arrangement is shown in Table 2. In the case of a horizontal arrangement, bins are no longer represented as a set of tuples but rather as different columns (cf. Table 3). One obvious advantage of the horizontal layout over the vertical is the reduced redundancy.

imageid	**x**	**y**	red	green	blue	grey
...	...	...	...	...	...	...

Table 1: Schema of RGB_Grey

id	**chan.**	**bin#**	**min**	**max**	bin	freq.
...	...	256	...	...	0	...
...	...	256	...	...	...	...
...	...	256	...	...	255	...

Table 2: Vertical histogram schema

id	**chan.**	**bin#**	**min**	**max**	bin_0	...	bin_255
...	...	...	...	...	...	...	...

Table 3: Horizontal histogram schema

Operation	Column Store	pgSQL
Load (rgb)	UDF	-
Load (rgbgrey)	UDF	-
Load (grey)	UDF	-
Convert (rgb to rgbgrey)	UDF/SQL	-
Convert (rgb to grey)	UDF/SQL	-
Filter rectangle	SQL	SQL
Filter circle	SQL	SQL
Calc. histogram_v	UDF/SQL	SQL
Calc. histogram_h	UDF/SQL	SQL
Average histogram_v	SQL	SQL
Average histogram_h	SQL	SQL
Euclidean histogram_v	SQL	SQL
Euclidean histogram_h	SQL	SQL
Manhattan histogram_v	SQL	SQL
Manhattan histogram_h	SQL	SQL

Table 4: Operation implementations

3.2 Operation Definition

Processing images requires a set of operations as presented in the basic cIPT architecture. In an application, histograms might be compared to each other or to a previously created average histogram. Operations for these kinds of workflows are provided within the cIPT. They are implemented in either C++ and exposed to DBMS as User Defined Functions (UDF) or as SQL statements. Table 4 presents the existing functionality of cIPT for the commercial column-store and PostgreSQL. The parameters of UDFs and SQL implementations are equal. Their performance is evaluated in the following Section 4.

4 Experimental Evaluation

The evaluation of the cIPT is based on multiple defined test scenarios. Each of them is executed on the same system with same *original* dataset of the 100

[4] Similar layouts exist for the *rgb* and *grey* relation

images from the Cape2Cape challenge. A few of the most interesting results are listed below.

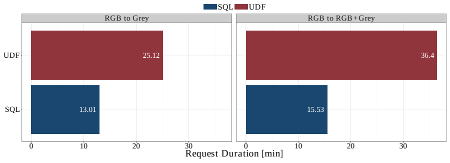

Fig. 2: Execution times of conversions

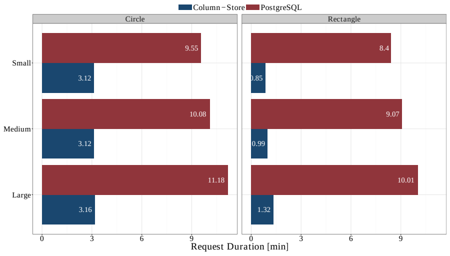

Fig. 4: Filtering on different databases

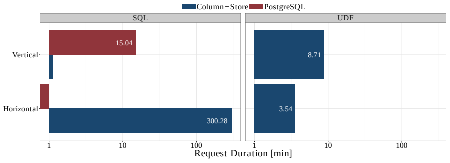

Fig. 3: Execution times of different oriented histogram calculations

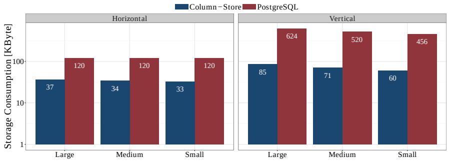

Fig. 5: Space consumption of histograms

Is the compression factor of a column store equally effective as the one of a row store?
To test this research question the *original* data set is loaded into the column store and PostgreSQL database. After the execution the space consumption of all relations are reported. Comparing these measurements clearly shows that the space consumption in the row store is dramatically worse. This is because PostgreSQL requires, even though no index is defined, space for tuple and page header. In addition it reserves about 10% to 20% of each page's available space to increase the runtime of updates at a later time.

Is the calculation of missing information more reasonable during the initial load or with a further processing step, utilized either with an UDF or via SQL?
Additional information can be calculated on the fly throughout the load- or a further processing-step. This experiment measures representative the execution times of UDFs and SQLs calculating the missing grey layer and compares these with a full load of the rgbgrey representation. Figure 2 demonstrates that an SQL implementation is with a factor of 0,42 to 0,51 dramatically faster than the UDFs explainable by the closer operation of SQL within the database's layout. However, UDFs ability to express complex custom implementation is not negligible. Regarding the postponed calculation of information at all, the clear suggestion to compute as much information as possible during the load can be provided. Every further processing step has to read the entire data. Yet, the load

operation has this data present as well during its execution and avoids unnecessary reads.

Which is the fastest filtering option on both databases?
Different filter operations are investigated within this test scenario. Therefore the *original* dataset provides the base data on which rectangles and circles are extracted in various sizes by SQL implementations on the column and the row store. Figure 4 presents the resulting execution times of every filter operation. Obviously recognisable is the significant difference between the both database types. The column store takes only up to 10% of the time the row-oriented database consumes. This clear advantage might scale with increasing data. Moreover, filtering less complex shapes like rectangles improves the run time of these operations as well. The correlation between execution time and filtered size is less a influence factor and can be broken down to increased data transfer.

Which database extracts features more efficiently, either with a UDF or SQL?
This test scenario measures the duration of cIPT's feature extraction operation on both database types. Therefore a UDF and SQL implementation creates vertical and horizontal histograms of the *original* dataset's images. In Figure 3 the constant duration of UDFs with 4-9 minutes is recognisable. The results of the SQL implementation are very wide spread. After analysing the biggest outlier with more than 300 minutes reveals that the multiple DECODE statements are highly inefficient. Using a UDF to transpose the result of a vertical histogram operation as a work-around decreases the execution time dramatically. The remaining measurements conclude that the column-store performs better on creating histograms both with SQL and UDFs.

What is the space consumption with different histogram orientation types on a row and a column-store?
The last test scenario analysis the space consumption of histograms with different orientations and image areas as input. Figure 5 depicts the measurements for every type on both database types. The correlation between the input size and space consumption can be explained by the implementation. Since the probability that a bin becomes assigned decreases with the shrinking image area, the amount of unassigned bins increases. These bins can either be compressed very efficiently as their value is 0 or there are non-existing e.g. vertical layout. While the commercial column-store profits by increasing the compression rate, PostgreSQL with its fixed size per cell is limited to the 120 KBytes in space consumption. The most efficient combination is as expected the horizontal layout on the column-store because the layout adapts the characteristics of the column-store and can compress the data to tiny 33-37 KBytes.

5 Conclusion

The development of the cIPT yield a first prototype of an image processing feature for column oriented databases. Evaluating the included different-oriented data layouts on PostgreSQL as a row-store and a commercial column-store reveals the advantages and disadvantages of each. Furthermore, the much better compression of the column-store is clearly visible in both, the loaded image data and calculated features.

Furthermore, experiments with the cIPT's operations analyse the differences between implantations of proprietary UDFs and queries using SQL. UDFs are constructed for very flexible use cases and complex calculation which could not be put into practise via SQL. Especially the loading process of the cIPT is realised with a special UDL, implementing several open-source C++ libraries for image processing. However, queries in SQL are significantly faster since they operate much closer on the database system and are efficiently restructured by the database's build-in query optimizer.

Using cIPT for a real world scenario, classifying recorded images during a road trip from the north to the south cape by a mounted camera shows one possible application. This responsible classifier analysed the red and blue colour values of 4000 images within special areas. Unfortunately, the results are premature and point out further improvements for such an application of the cIPT.

References

1. Abadi et al. Column-oriented database systems. *VLDB*, Aug. 2009.
2. R. Datta et al. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.*, 40(2):5:1–5:60, May 2008.
3. L. Deligiannidis and H. Arabnia. *Emerging Trends in Image Processing, Computer Vision and Pattern Recognition.* Emerging Trends in Computer Science and Applied Computing. Elsevier Science, 2014.
4. HP. Cape2cape. `http://www8.hp.com/uk/en/campaigns/cape2cape/overview.html`.
5. M. Ivanova et al. Monetdb/sql meets skyserver: the challenges of a scientific database. In *In Proc. SSBDM 2007.*
6. B. Johansson. A survey on: Contents based search in image databases. Survey, Department of Electrical Engineering, Linköping University, 08 2000.
7. A. Lamb et al. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, Aug. 2012.
8. Niblack et al. Qbic project: querying images by content, using color, texture, and shape, 1993.
9. L. Sidirourgos et al. Column-store support for rdf data management: Not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, Aug. 2008.
10. M. Stonebraker et al. C-store: A column-oriented dbms. In *In Proc. VLDB*, 2005.