Datenbank-Spektrum (2016) 16:17–26
DOI 10.1007/s13222-015-0209-y

SCHWERPUNKTBEITRAG

ADAM$_{pro}$: Database Support for Big Multimedia Retrieval

Ivan Giangreco · Heiko Schuldt

Received: 13 October 2015 / Accepted: 20 December 2015 / Published online: 29 January 2016

Abstract For supporting retrieval tasks within large multimedia collections, not only the sheer size of data but also the complexity of data and their associated metadata pose a challenge. Applications that have to deal with big multimedia collections need to manage the volume of data and to effectively and efficiently search within these data. When providing similarity search, a multimedia retrieval system has to consider the actual multimedia content, the corresponding structured metadata (e.g., content author, creation date, etc.) and—for providing similarity queries—the extracted low-level features stored as densely populated high-dimensional feature vectors. In this paper, we present ADAM$_{pro}$, a combined database and information retrieval system that is particularly tailored to big multimedia collections. ADAM$_{pro}$ follows a modular architecture for storing structured metadata, as well as the extracted feature vectors and it provides various index structures, i.e., Locality-Sensitive Hashing, Spectral Hashing, and the VA-File, for a fast retrieval in the context of a similarity search. Since similarity queries are often long-running, ADAM$_{pro}$ supports progressive queries that provide the user with streaming result lists by returning (possibly imprecise) results as soon as they become available. We provide the results of an evaluation of ADAM$_{pro}$ on the basis of several collection sizes up to 50 million entries and feature vectors with different numbers of dimensions.

Keywords Databases · Multimedia retrieval systems · Big data

I. Giangreco (✉) · H. Schuldt
Department of Mathematics and Computer Science,
University of Basel, Basel, Switzerland
e-mail: ivan.giangreco@unibas.ch

1 Introduction

Multimedia is *Big Data*, both in terms of their volume and their heterogeneity. Many applications that have to deal with such big multimedia collections need support for managing the sheer volume of data and for effectively and efficiently searching within these data—based on annotated (structured) metadata and/or based on intrinsic features of the multimedia objects. Consider, for instance, the following applications: a TV station is looking for videos and scenes with some specific visual content to enrich a news report; a medical researcher is looking for all mammograms showing a certain visual characteristic which might indicate a case of breast cancer; or a user is looking for a piece of music that she remembers without, however, knowing its name.

Obviously, all these applications have in common that large multimedia collections need to be searched on the basis of their content. In the last years, this has successfully spurred research in the field of machine learning in order to detect and possibly learn features for characterising the content of multimedia objects and thus to serve as basis for retrieving results when comparing the object features to a query (e.g., [9, 10]). However, these feature extractors only form one side of the coin. The other side of the coin is formed by the organisation and storage of feature data (in general: any form of metadata on multimedia objects), and the support for various types of queries using these metadata.

In this paper, we address multimedia queries that combine Boolean retrieval based on structured metadata (e.g., content author, creation date, etc.) with the vector space retrieval model to support similarity queries on the objects' content. By this, we consider both the information retrieval and the database approach: Information retrieval systems traditionally focus on high-dimensional feature spaces and support nearest neighbour queries (similarity search). Database systems, on the other hand, traditionally only come with very limited support for similarity searches and partial matches, but are very good at organising data and retrieving exact matches. For multimedia content and its associated metadata, the 'one size fits all' approach generally does not work. As a consequence, multimedia retrieval systems often store feature data in file-based structures. From a database perspective, this has various drawbacks: On the one hand, the absence of *physical* and *logical data independence* makes the organisation of data a difficult and tedious task that is prone to errors. On the other hand, by letting the retrieval system take care of storing data, the principle of *separation of concerns* is violated in that application logic (i.e., *what* feature data to store) and storage logic (i.e., *how* to organise and store feature data) are not well separated. The missing separation of concerns leads to the re-engineering of components necessary for the sole task of managing and organising data.

In this paper, we introduce ADAM$_{pro}$, a database and information retrieval system that jointly supports Boolean and vector space retrieval and that is particularly tailored to very large multimedia collections. ADAM$_{pro}$ is an extension of the ADAM system [6, 7] and focuses on storage support for big multimedia data. It follows a modular architecture: based on the nature of the data to be managed (structured or unstructured), individual modules can be replaced to increase the overall query efficiency and to reduce response time. In addition, ADAM$_{pro}$ jointly supports various index structures. By combining index structures that quickly produce (approximate) results with index structures that take longer to produce (correct) results, so-called *progressive* queries can be supported. In the context of the retrieval process, results are presented to a user in a streaming fashion as soon as they become available. The query results will be continuously updated as more (and also more precise) results will be available. While this is not necessary for small collections, *pro*gressive queries in ADAM$_{pro}$ allow to provide fast query results especially for big multimedia collections.

The contribution of the paper is threefold:

1. We present the architecture of the ADAM$_{pro}$ system and the interplay between its components at query time to be able to efficiently process a query. In this paper, we focus on the vector space retrieval model in which feature data reflecting the objects' content is represented by means of (high-dimensional) feature vectors and a k nearest neighbour (kNN) search is applied to find the most similar objects.
2. We introduce the concept of progressive query evaluation which aims at reducing retrieval time. ADAM$_{pro}$ seamlessly combines various index structures, i.e., Locality-Sensitive Hashing (LSH), Spectral Hashing, and the Vector Approximation File (VA-File), to answer kNN searches efficiently; the former two are used to generate early, albeit pre-mature (approximate) results, while the latter guarantees exact results, but at the price of higher retrieval times.
3. We present results of the evaluation of ADAM$_{pro}$ under varying configurations and particularly address the scalability of ADAM$_{pro}$. This includes both the sheer size of collections (no. of objects) as well as the complexity of the feature data (no. of dimensions of the feature space).

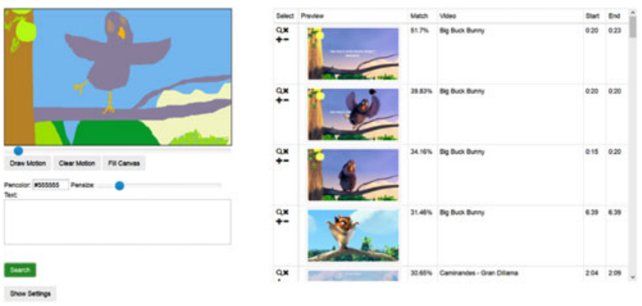
The remainder of the paper is organised as follows: in Section 2, we present a sample application. Section 3 discusses details of the system architecture of ADAM$_{pro}$ and introduces the concept of progressive queries. Section 4 reports on the evaluation of ADAM$_{pro}$. Section 5 discusses related work and Section 6 concludes.

2 IMOTION: A Sample Application

As an example for the use of ADAM$_{pro}$, consider IMOTION (Intelligent Multi-Modal Augmented Video Motion Retrieval System)[1], a system for large-scale video retrieval applications [14]. The objective of IMOTION is to provide a rich variety of different query paradigms for searching in video collections. This includes traditional *keyword search* on the basis of automatically collected metadata (e.g., content author, video length, etc.) or manually added tags describing the content of a video or a shot. In addition, IMOTION also supports a large variety of similarity search-based queries, e.g., *Query-by-Sketch (QbS)*, *Query-by-Example (QbE)*, querying by motion and querying by audio. For the retrieval, this means that users can specify either an existing image or a video snippet as a query, or provide the system with a hand-drawn sketch of the most relevant object(s) (e.g., using a color sketch as depicted in Fig. 1), or draw motion or record an audio snippet to search for. All the query options can be seamlessly combined—either in a pipelined fashion (e.g., start with a keyword query and use one of the results as query object for QbE), or combined in a single query by superimposing for instance a sketch and a query image (by adding, via a sketch, an object that is not visible on the query image).

For supporting retrieval, IMOTION makes use of a large number of feature extractors [14] which produce comparatively high-dimensional dense feature vectors (up to over several hundred dimensions) that are used at query time for comparison to the query object. Given the large size of the video collections, which the IMOTION query engine is supposed to handle, this exceeds the capabilities of current storage systems that are not adapted to the use case at hand (as will be shown in Sect. 4). In the IMOTION system, ADAM$_{pro}$

[1]http://www.imotion-project.eu



Select	Preview	Match	Video	Start	End
		51.7%	Big Buck Bunny	0:20	0:23
		39.83%	Big Buck Bunny	0:20	0:20
		34.16%	Big Buck Bunny	0:15	0:20
		31.46%	Big Buck Bunny	6:39	6:39
		30.65%	Caminandes - Gran Dillama	2:04	2:09

Fig. 1 Sketch-based video retrieval in IMOTION (from [14])

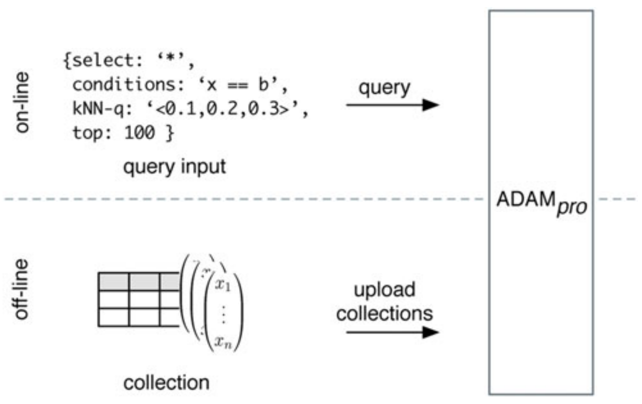


Fig. 2 On-line and off-line phase of a retrieval system

takes over the task of storing and organising both the structured metadata and the extracted feature vectors, and supports the retrieval logic for retrieving exact or approximate matches.

3 Architecture

In retrieval systems such as the IMOTION system, two phases can generally be distinguished: In the *off-line* phase, the retrieval engine extracts features from the given multimedia objects. The feature extraction phase serves two purposes: First, it allows the adaption of a (multimedia) document to the retrieval framework used; second, it reduces the search complexity, since it avoids the full inspection of objects, but only considers the extracted features for comparison. In the *on-line* phase, i.e., the time-sensitive query phase, on the other hand, the extracted features are compared to the query object, and ranked by similarity.

Figure 2 shows ADAM$_{pro}$ in the described setting: in the off-line phase, ADAM$_{pro}$ takes over the task of storing the feature vectors as they are inserted into the system. In the on-line phase, ADAM$_{pro}$ is responsible for a fast and precise response given a query.

To this end, ADAM$_{pro}$ combines various storage subsystems depending on the data and the queries at hand; the combination of various systems shows the advantage of each system over the others (and over a monolithic system) in one specific phase of the retrieval:

- for structured metadata, ADAM$_{pro}$ uses a relational DBMS, as it allows querying for all attributes in a very elegant manner with good performance; these data are used for a pre-filtering,
- the index structures for a k nearest neighbour (kNN) search are stored in a file-based format, as it can be well distributed over multiple workers; these data are used to further filter the tuples to retrieve,
- for the feature vector data, we use a key-value store which allows to retrieve the full feature vectors (using the keys filtered by the index) quickly and efficiently for further computation; these data are used for the full distance computation.

The overall architecture of ADAM$_{pro}$ is depicted in Fig. 3: On the left hand side, the orchestrator is depicted, which takes care of incoming requests (such as insert operations or queries) by calling the corresponding components. The *metadata storage component* is responsible for storing and retrieving structured metadata in a relational database. The *index storage component* builds the index structures and stores them in index files, separated from the actual content and metadata. Finally, the *feature storage component* stores the full feature vectors. At query time, the metadata storage component will filter results based on Boolean predicates (i.e., on the metadata, for instance `date = 15/03/2015`). The results will then be processed by the index storage component that further prunes the result list by using the index structures available for retrieving the k nearest neighbours and performing a similarity search on the basis of the given query vector. Finally, the remaining elements are collected from the feature storage component that retrieves the full feature vectors and performs the exact computation of distances.

3.1 Schema Definition and Data Insertion

In the data definition step, the user specifies the logical schema for the data. Given the various data types, ADAM$_{pro}$ distributes the physical data to different subsystems: ADAM$_{pro}$ stores feature data in a key-value store and indexes the data using various index structures built into the system. The metadata, on the other hand, is stored in a relational DBMS. In this phase, ADAM$_{pro}$ takes care of creating the appropriate schema in the various systems involved.

3.2 Retrieval

Consider as an example the following query: A user is looking for the top 100 images that were taken on the 15th of

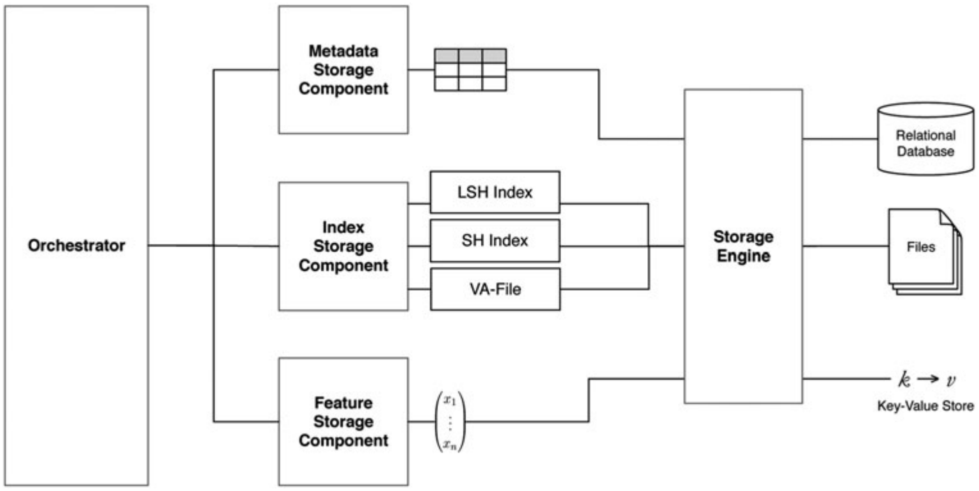


Fig. 3 Architecture of the ADAM$_{pro}$ system

March 2015 and that are similar to the given query image. This query uses a Boolean predicate (i.e., `date = 15/03/2015`) and involves a similarity query (i.e., all objects similar to the query image or all feature vectors similar to the query vector, respectively). The results should obviously be ordered by similarity and pruned at 100 results. As shown in this example, a query in this setting may ask for all objects similar to the given query object while at the same time fulfilling all Boolean predicates.

ADAM$_{pro}$ supports both Boolean retrieval and kNN similarity search. Retrieval based on Boolean predicates can be applied on all structured fields (i.e., the structured metadata). The similarity search is performed using the feature vectors. To increase the retrieval performance, various index structures for high-dimensional data are used (Sect. 3.3) that prune results from the final result set (as they take into account the limiting factor k of a kNN search), rather than only ordering the results. Therefore, it is crucial that the Boolean retrieval is always performed before the kNN search, as otherwise results would get lost. Furthermore, by first performing the (fast) Boolean retrieval, the similarity search can avoid to consider results that do not adhere to the Boolean predicate and by that improve the system's performance.

In Fig. 4, we show a high-level execution plan for a query in the ADAM$_{pro}$ system: (1) If Boolean predicates are available in the query, the query is first sent to the relational database that returns a result set fulfilling the Boolean predicates (TID list). (2) Using the built-in index structures, the query vector and the TID result set from step 1, in the second step, the nearest result candidates are retrieved. This result set contains possibly more than k elements and is not yet sorted. Furthermore, the result elements do not yet contain the exact distance values. (3) In the last step, using the TID

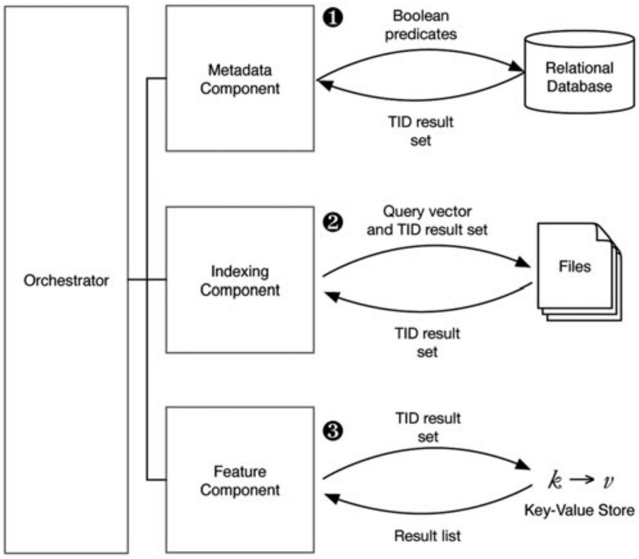


Fig. 4 General query plan in the ADAM$_{pro}$ system

set of step 2, the full feature vectors are retrieved and the exact distances are computed. The results are then returned.

3.3 Index Structures

To support efficient k nearest neighbour retrieval, the ADAM$_{pro}$ system implements Locality-Sensitive Hashing (LSH) [8], Spectral Hashing (SH) [18] and the Vector Approximation-File (VA-File) [17]. We detail the implemented index structures in the following.

3.3.1 Locality-Sensitive Hashing

The main idea of Locality-Sensitive Hashing [8] is to hash each feature vector f using several hash functions and use the hashes for finding near neighbours. Formally, using a hash function, a d dimensional vector is mapped onto the space of integers ($h_a(f) = \mathbb{R}^d \mapsto \mathbb{N}$). A family $\mathcal{H}$ of hash functions is called locality-sensitive or more specifically (R, cR, P_1, P_2)-sensitive, if for any two feature vectors $p, q \in \mathbb{R}^d$:

- if $||p - q|| \leq R$ then $P_{\mathcal{H}}[h(q) = h(p)] \geq P_1$
- if $||p - q|| \geq cR$ then $P_{\mathcal{H}}[h(q) = h(p)] \leq P_2$

with $P_1 > P_2$. Intuitively, if two vectors are close, the probability that they collide in their hash should be high; vice versa if two vectors are far apart, the probability that they collide in their hash should be small.

ADAM$_{pro}$ currently supports the Minkowski distances; therefore, for LSH, we use the family of hash functions proposed in [4] for l_p norms, based on p-stable distributions. For the hash function, we pick a random projection $a \in \mathbb{R}^d$ with entries from a p-stable distribution, chop the line into equi-width segments (w) and shift by a random value $b \in [0, w)$. Formally, the hashing function is given as $h_{a,b}(v) = \lfloor \frac{av+b}{w} \rfloor$.

While LSH can be very efficient, it has to be noted that the hashing approach yields both false positives (irrelevant items in the result list) and false negatives (missing relevant items).

3.3.2 Spectral Hashing

Spectral Hashing [18] belongs also to the family of hash-based indexing methods, however, falls into the category of ‘learning to hash’, i.e., the hash functions are generated based on the data at hand. The idea of Spectral Hashing is to find a hash function such that similar items are mapped to similar hash codes, i.e., small distances in the feature space should result in small Hamming distances between the codes. The embedding is done using the eigenfunctions computed along the principal component analysis (PCA) directions. In essence, the algorithm will

1. find the principal components of the data using PCA
2. compute the Laplacian eigenfunctions with the smallest eigenvalues along every PCA direction
3. threshold the eigenfunctions at zero to obtain the binary codes

As with LSH, Spectral Hashing may yield both false positives and false negatives.

3.3.3 Vector Approximation (VA) File

Behind Vector Approximation-File (VA-File) [17] lies the idea to build an index that yields exact results and at the same time is very performant. The authors argue that for increasing dimensionality, any tree-based index structure degenerates to a sequential scan. Therefore, the authors suggest to compress the feature vectors in a quantization step to a short signature that can quickly be scanned in a sequential manner and which allows to early prune the result list. This is achieved by using the signatures to compute at query time upper and lower bounds to the distance and by that early exclude items that are too distant. The upper and the lower bound of the distance can be calculated with very few simple calculations and the computation is therefore computationally less complex than a full distant computation on the vectors. Furthermore, by only reading the signatures less page accesses and therefore I/O accesses are necessary, which would largely increase the retrieval time. While the VA-File may degenerate to a sequential scan, it will always return all true positives.

To create a VA signature, a fixed-length bit string for each data point is generated. For that purpose, the data space is divided in $2^{b_{tot}}$ cells, where b_{tot} denotes the total length of the bit signature, and the cells are enumerated in a binary way. Each dimension d receives b_d bits that are finally concatenated to create the full bit mask.

3.4 Progressive Query Results

Queries in multimedia retrieval systems are often comparably long-running queries, as they cannot profit from tree-based index structures that significantly reduce the search time complexity. The reason for this is that feature vectors, on which the queries are performed, do not have an absolute ordering, but the ordering of results is only based on the given query. [17] argues that with increasing dimensionality, tree-based index structures degenerate to a sequential scan of the data. Therefore, predicting the query time for indexes for high-dimensional data is a very difficult task: Traditional database systems consider for this the number of index and data pages, the height of the index tree, the length of TID lists in the leaf nodes of the tree, etc. For the indexes used in ADAM$_{pro}$, these parameters are not appropriate predictors for the retrieval time. Consider, for example, a VA-File index: The algorithm behind the VA-File will scan all database elements. However, by decreasing the size of the signature to be scanned, the number of operations to be performed for computing the final distance and consequently the number of data pages to be loaded can be significantly decreased. Nevertheless, using the number of index pages containing the signature for estimating the retrieval time will actually not correctly estimate the retrieval time, as in the most degenerate case, the VA-File has to consider all vectors stored on the data pages. In particular, as this is not an inherent property of the data only, but of the data in combination with the query at hand, it is a hard task to predict the query time. Finally, these predictions do not consider whether the indexes will return exact or only approximate results.

For this reason, ADAM$_{pro}$ supports so-called *progressive querying* that results in streamed result lists. Using this approach, ADAM$_{pro}$ runs at the same time all physical plans to execute the same logical plan and returns the (possibly approximate) answer(s) as soon as they become available to the user. Starting the same query using different query plans at the same time may decrease the efficiency of the entire system, but it allows to trade computation (which is not a bottleneck in modern environments) with query response time. For a user who is waiting a short time for her results, the results from the database may only be very approximate; if the user, on the other hand, waits a bit longer, she may get an answer that is more precise. In any case, she will get the first possible answer as soon as it is available.

Consider, for clarification, the following example: A query for the k nearest neighbours is started on all available indexes: the Spectral Hashing index may return first, due to its low query complexity, however the result may contain false positives or lack true positives. Only in the next step, the exact results from the VA-File index may arrive. On the other hand, in very degenerate cases, a sequential scan may return even before the VA-File index, ensuring that the user gets the results as fast as possible.

3.5 Distribution

In a distributed setting, there is no obvious partitioning scheme that can be generally applied to the feature data and that allows to prune nodes at query time from the retrieval without possibly loosing result elements (this is also a consequence of the fact that tree-based methods do not work well for feature vector data). While, for instance, text retrieval systems can choose to query only specific nodes that are responsible for a certain keyword appearing in the query, this approach is not easily adaptable to kNN retrieval in the multimedia context. As a consequence, for a query, all data items have to be considered, i.e., a query has to be processed by all nodes and the sub-results of each node have finally to be merged. The distribution of the VA-File, for instance, has already been discussed in [16]. We have implemented the same ideas for all index structures and we perform the retrieval in ADAM$_{pro}$ in a map/reduce fashion. At query time, the index structures are partitioned to be processed by multiple workers (possibly residing on the same node). Each node takes care of filtering out the k nearest neighbours and sends the partial result list to the master node that merges the local result lists to a global result list of nearest neighbours.

4 Performance Evaluation

4.1 Implementation

ADAM$_{pro}$ is implemented in Java/Scala using Apache Spark 1.5[2]. For storing the metadata, we use PostgreSQL[3], the Apache Parquet columnar file format on the Hadoop file system (HDFS) for the indexes, and Apache Cassandra 2.1[4] as key-value store for storing the feature vectors. ADAM$_{pro}$ can be accessed via a REST interface. LSH is implemented based on E2LSH[5] and Spectral Hashing is based on the Matlab code provided by the authors[6].

4.2 Experimental Setup

We have evaluated ADAM$_{pro}$ using artificially generated data at various numbers of dimensions and collection sizes. The evaluation setup involves the following parameters:

- collection size: 10K, 50K, 100K, 500K, 1M, 5M, 10M, 20M, 50M
- dimensions: 10, 50, 100, 200, 500
- execution plans: sequential scan, LSH scan, Spectral Hashing scan, VA-File scan

The vectors added to the collection and the query vectors are composed of uniformly distributed float values $\in (0, 1)$. To avoid anomalies in the results, we have run each experimental setting five times and average over the different runs for the same parameter setting. We run ADAM$_{pro}$ (and the systems to compare it to) on Microsoft Windows Azure using one DS13 instance running Ubuntu 14.04 with 8 cores and 56 GB memory. For all experiments, the parameters for generating the index (number of bits for signature, etc.) have been set beforehand to a general value independent of the given data.

As a baseline, we use PostgreSQL 9.4[7] and MongoDB 3.0[8]. In PostgreSQL, we use a custom function to compute the distance between two float arrays and use `ORDER BY` and `LIMIT` to model a k nearest neighbour search. In MongoDB, on the other hand, we make use of a server-based script that computes in a map/reduce fashion the distance between a query array and the vectors stored in the collection: the map function emits the distance between the query vector and the vector from the collection, whereas the reduce function sorts and slices the results. Both baselines do not make use of any index structure that could improve the kNN retrieval.

[2]http://spark.apache.org
[3]http://www.postgresql.org
[4]http://cassandra.apache.org
[5]http://www.mit.edu/∼andoni/LSH
[6]http://www.cs.huji.ac.il/∼yweiss/SpectralHashing
[7]http://www.postgresql.org
[8]http://www.mongodb.com

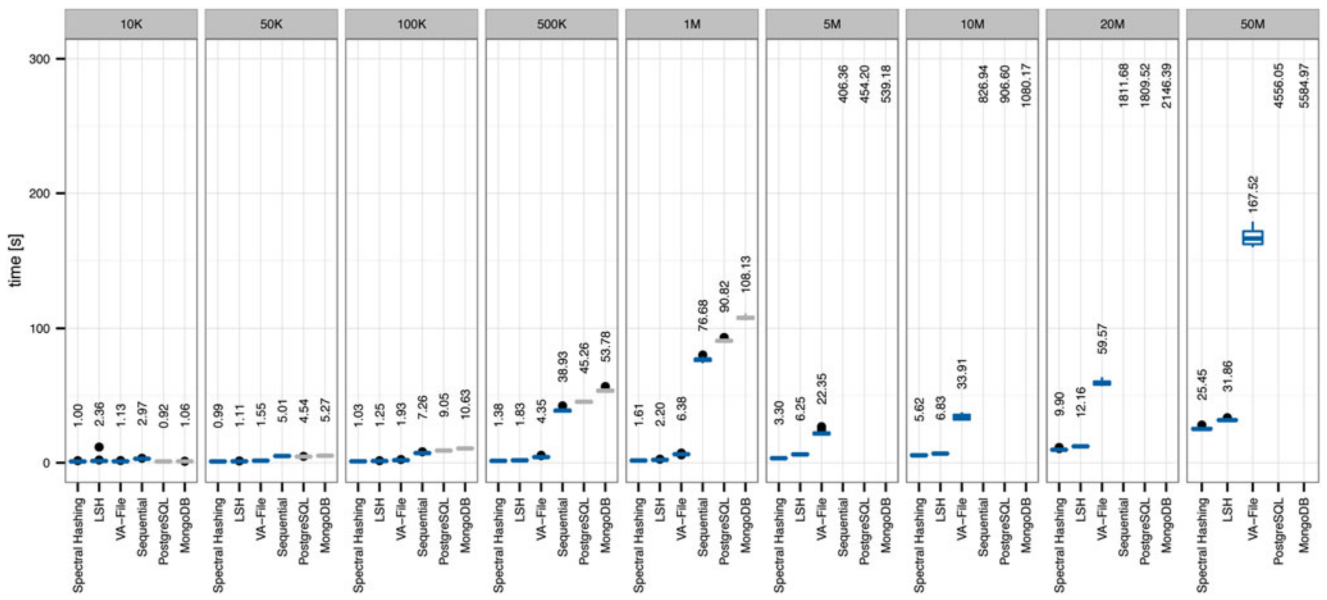


Fig. 5 Performance evaluation showing the retrieval time at varying collection sizes for the various methods implemented in ADAM$_{pro}$ with dimensionality of the feature vectors at 100. As a baseline, both PostgreSQL and MongoDB have been added to the plot. The experimental runs are summarised by the mean time for each method. The plot is cut at 300 s; for all values above 300 s only the mean time is displayed

4.3 Evaluation of Single Execution Plans

We first consider the results of the evaluation for every single execution plan: Fig. 5 shows a box plot that compares the retrieval time at varying collection sizes with a fixed dimension of 100 for the feature vectors. Note that the plot has been cut at 300 s and for all values above this threshold, only the mean time is displayed. It can be seen that when increasing the number of items, Locality Sensitive Hashing and Spectral Hashing retrieve the results faster than the other scanning methods (i.e., than the VA-File scan and the sequential scan). Furthermore, it can be seen that in our experiments the VA-File always performs better than the sequential scan. This means that given the data and the queries used in the evaluation, we never get into the degenerate case in which a sequential scan is truly necessary.

This behaviour is as expected, as LSH and SH allow for a simple lookup, whereas the VA-File has to scan all signatures; however, in exchange, the VA-File returns precise results. For increasing collection size, in particular for collections that contain more than 50K elements, our system performs in any case better than performing a sequential scan in MongoDB or PostgreSQL, the baseline to our evaluation.

As can be seen from Fig. 5, the retrieval time is obviously dependent on the collection size. Moreover, as it can be seen from Fig. 6, it also depends on the dimensionality. Figure 6 shows the retrieval time for increasing numbers of dimensions of the feature vectors (at a fixed collection size of 1M elements). As can be seen, the retrieval time increases with the dimensionality of the feature vector.

4.4 Evaluation of Progressive Querying

Given these observations, we evaluate the behaviour of progressive querying. In particular, we show the results exemplified at collection sizes of 100K (Fig. 7a) and 1M (Fig. 7b) elements, respectively. In Fig. 7, we show at which time after starting the query ADAM$_{pro}$ presents its results to the user. In the original ADAM$_{pro}$ implementation, the query execution is normally cancelled as soon as exact results (i.e., results from the sequential scan or from VA-File) are retrieved; for this evaluation, to be able to show the times of the various scans, we have adjusted the implementation not to abort the execution (nevertheless, we have marked the time at which ADAM$_{pro}$ would stop the further execution in its normal setup with a red line). This means that even after finding the final and exact results, we continue to execute the query and measure the query time for the remaining execution plans.

In Fig. 7, it can be seen that predicting the query time for the various index structures is a difficult task: it cannot be clearly stated which index structure performs better under which conditions. Particularly for small dimensionalities, it is not obvious whether LSH, Spectral Hashing or the VA-File will return its results first. With increasing dimensionality, both hashing based methods perform clearly better than the VA-File and the sequential scan. Particularly for in-

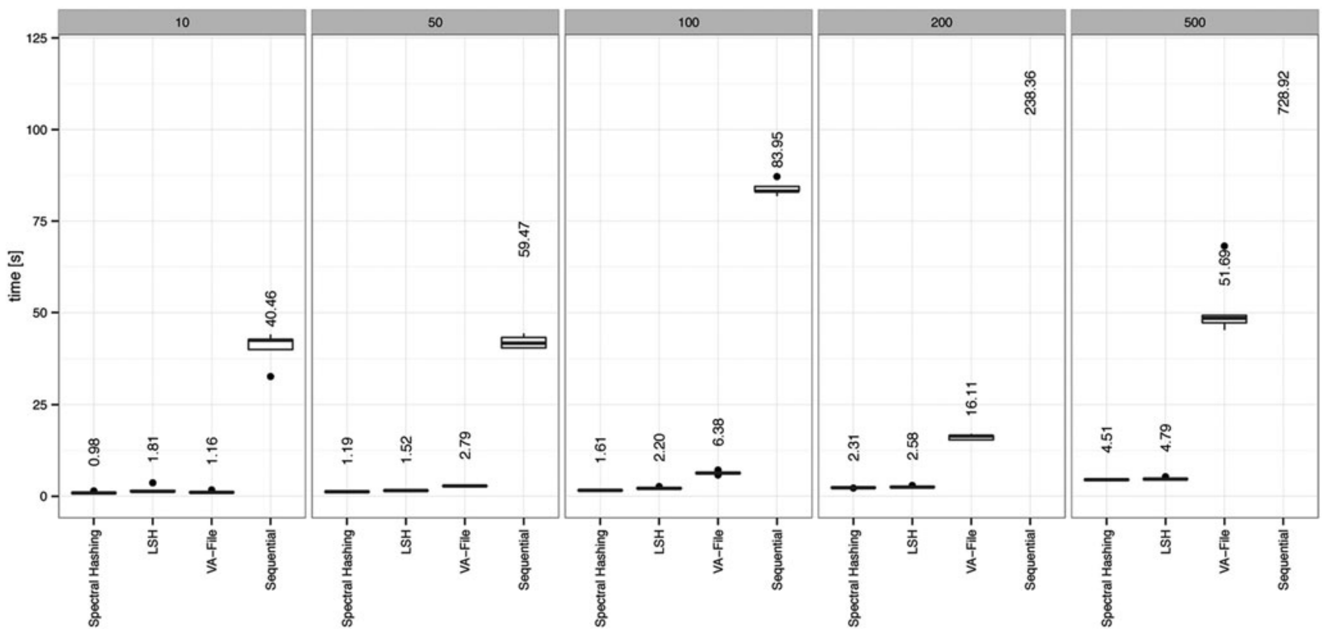
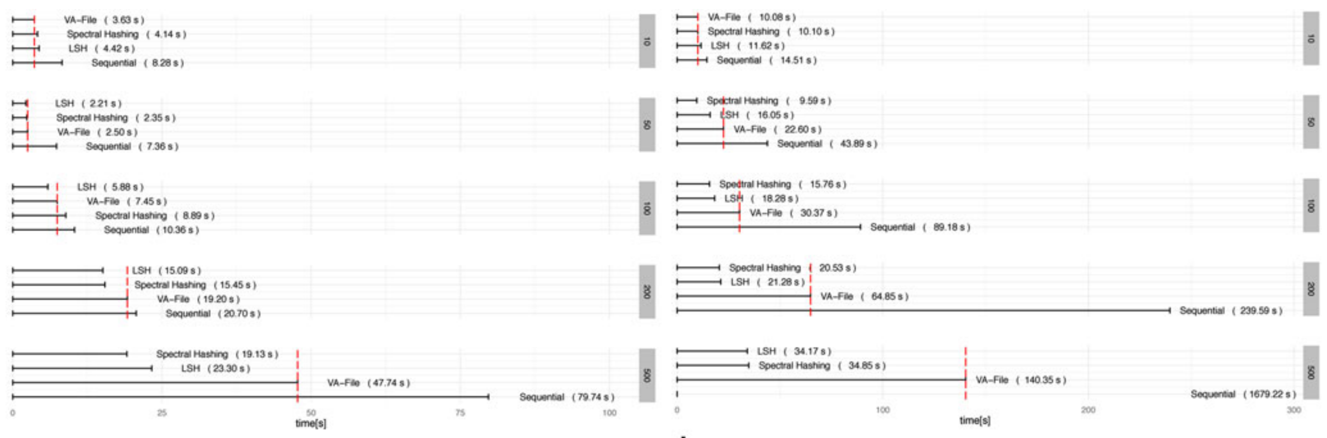


Fig. 6 Performance evaluation showing the retrieval time at varying number of dimensions of the feature vectors stored in ADAM$_{pro}$ using the various methods implemented at a collection size of 1M elements. The plot is again cut at 120 s and for all values above this threshold only the mean time is displayed



a Collection with 100K elements. **b** Collection with 1M elements.

Fig. 7 Timeline displaying the mean time at which the results using the various scans become available when using progressive querying for various dimensions at a collection size of 1M. The red line denotes when ADAM$_{pro}$ would stop the further execution of the retrieval, as precise results have been found

creasing dimensionality (and collection size), the progressive querying approach becomes more and more important: If, in a collection of 1M elements and feature vectors with a dimensionality of 200 elements, a user realises that the results received after about 20 s are good enough or not worthy to consider further, she may as well abort the further execution of the query; on the other hand, she may wait to get precise results after about 60 s. As expected, in all runs, the sequential scan is the last execution plan that returns its query results. Since our query handler is aware of the fact that VA-File returns precise results, it would abort the further execution after VA-File has answered.

5 Related Work

One of the earliest works attempting to integrate an information retrieval system and a database management system can be found in [15]. In this early work from 1980, the authors note especially the lack of support for information retrieval queries in the query language and in the internal indexing techniques of a database management system (DBMS). The authors motivate the integration of the two systems into a database management and information retrieval system (DB-MIRS) by the need for queries supporting both formatted (structured) and unformatted (unstructured) data retrieval.

An early integration of multimedia data in databases can be found in the IBM project Garlic [3]. In here, the authors integrate multiple federated databases (some of those from the IBM project QBIC) into one distributed system for multimedia data. The system is based on an object-oriented database model. For query formulation, the authors extend the object-oriented query language (OQL).

Similarly, [12] introduces Chabot, an information retrieval system using an underlying PostgreSQL database for storing extracted features. The authors implement complex types, user-defined indexes and user-defined functions into the database to support information retrieval data types and queries. Chabot not only supports text-based queries, but also allows to improve results by providing content-based queries (e.g., “images with some orange in it”). Both Garlic and Chabot support Boolean predicates rather than similarity-based queries.

[5] presents Mirror, a database supporting content-based multimedia retrieval. The authors describe the engineering factors for creating a distributed multimedia IR-DBMS that uses Moa, a new relational algebraic framework based on the non-first normal form (NF^2). Mirror is implemented on top of the object-relational DBMS Monet [2].

Further, DISIMA DBMS [13] is a DBMS that allows to store syntactic features, i.e., color, shape, texture, and semantic features, i.e., real world objects or concepts, in an object-oriented data model. The system supports content-based searches and searches on image semantics. The authors implement an extended version of OQL for multimedia objects (MOQL) and VisualMOQL, a visual counterpart to MOQL. To increase the performance of the system, the authors use three-dimensional extendible hashing (3DEH) that allows to pre-filter images based, for instance, on the average color.

[1] introduces a system that combines low-level (syntactic) features with semantic features in a commercial object-relational database. The database is extended by several User Defined Types following the MPEG-7 standard descriptors, and operations implemented in PL/SQL, e.g., to evaluate similarity measures.

In [11], the authors make use of the map/reduce paradigm for querying large sets of image data in a cloud environment. The authors use an indexing method called extended Cluster Pruning (eCP) for indexing the feature data and port it to map/reduce on the Hadoop platform.

6 Conclusion

In this paper, we have presented ADAM$_{pro}$, a modular system that manages large multimedia collections. ADAM$_{pro}$ flexibly supports various storage systems and indexing structures (LSH, Spectral Hashing, VA-File) to increase the overall query efficiency and reduce response time. Furthermore, with ADAM$_{pro}$, we have presented the concept of progressive queries that embraces the idea of returning a result stream to the user: depending on how long the user waits, the system will refine the results and produce correct instead of only approximate results.

In our future work, we plan to consider different distribution scenarios for ADAM$_{pro}$ and we plan to further increase the collection size and dimensionality.

Acknowledgments This work was partly supported by the Swiss National Science Foundation in the context of the CHIST-ERA project IMOTION, contract no. 20CH21_151571. Furthermore, the authors would like to thank the reviewers for their time in reviewing our manuscript and for their helpful comments to improve this paper.

References

1. Alvez CE, Vecchietti AR (2010) Combining Semantic and Content Based Image Retrieval in ORDBMS. In: Rossitza Setchi, Ivan Jordanov, Robert J. Howlett and Lakhmi C. Jain (eds) Knowledge-Based and intelligent information and engineering Systems, 14th International Conference, KES 2010, Cardiff, UK, September 8–10, 2010, Proceedings, Part II, volume 6277 of Lecture notes in computer science. Springer, Berlin, pp 43–55
2. Boncz PA, Kersten ML (1994) Monet. An impressionist sketch of an advanced database system. In: In Proc. IEEE BIWIT workshop, San Sebastian, Spain
3. Carey MJ, Haas LM, Schwarz PM, Arya M, Cody WF, Fagin R, Flickner M, Luniewski A, Niblack W, Petkovic D, Thomas J II, Williams JH, Wimmers EL (1995) Towards heterogeneous multimedia information systems: The Garlic Approach. In: RIDEDOM 1995: International workshop on research issues in data engineering - Distributed object management. Taipei, Taiwan, pp 124–131
4. Datar M, Immorlica N, Indyk P, Mirrokni VS (2004) Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004, SCG ’04, pp 253–262
5. de Vries AP, Blanken HM (1998) Database technology and the management of multimedia data in the mirror project. In: Proc. SPIE. International Society for Optics and Photonics, vol 3527, pp 443–453
6. Giangreco I, Al Kabary I, Schuldt H (2014) ADAM - A database and information retrieval system for big multimedia collections. In: Proceedings of the 2014 IEEE International Congress on Big Data, Anchorage, AK, USA, June/July 2014. IEEE, pp 406–413
7. Giangreco I, Al Kabary I, Schuldt H (2014) ADAM : a system for jointly providing IR and database queries in large-scale multimedia retrieval. In: Proceedings of the 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR’14), Gold Coast, Australia, July 2014. ACM, pp 1257–1258

8. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the 13th Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, pp 604–613
9. Karpathy A, Li F-F (2014) Deep visual-semantic alignments for generating image descriptions. CoRR, abs/1412.2306
10. Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ (eds) Advances in neural information processing systems 25. Curran Associates, Inc., pp 1097–1105
11. Moise D, Shestakov D, Gudmundsson GT, Amsaleg L (2013) Indexing and searching 100 m images with map-reduce. In: International Conference on multimedia retrieval, ICMR'13, Dallas, TX, USA, April 16–19, 2013, pp 17–24
12. Ogle V, Stonebraker M (1995) Chabot: Retrieval from a relational database of images. Computer 28(9):40–48
13. Oria V, Tamer Özsu M, Iglinski P (2001) Querying images in the DISIMA DBMS. In: MIS 2001: workshop on multimedia information systems, Capri, Italy, pp 89–98
14. Rossetto L, Giangreco I, Schuldt H, Dupont S, Seddati O, Sezgin M, Sahillioğlu Y (2015) IMOTION - a content-based video retrieval engine. In: Proceedings of the 21st International Conference on multimedia modeling (MMM'15), Part II, Springer, Sydney, Australia, January 2015, pp 255–260
15. Schek H-J (1980) Methods for the administration of textual data in database systems. In: Oddy RN, Robertson SE, van Rijsbergen CJ, Williams PW (eds) SIGIR 1980: International Conference on Research and development in information retrieval, Cambridge, England. Butterworth & Co, pp 218–235
16. Weber R, Böhm K, Schek H-J (2000) Interactive-time similarity search for large image collections using parallel VA-files. In: Research and advanced technology for digital libraries, 4th European Conference, ECDL 2000, Lisbon, Portugal, September 18–20, 2000, Proceedings, pp 83–92
17. Weber R, Schek H-J, Blott S (1998) A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB 1998: International Conference on very large data bases, New York, USA, pp 194–205
18. Weiss Y, Torralba A, Fergus R (2008) Spectral hashing. In: Advances in neural information processing systems 21, Proceedings of the Twenty-Second Annual Conference on neural information processing systems, Vancouver, British Columbia, Canada, December 8–11 , pp 1753–1760