

# Novel multimedia retrieval technique: progressive query (why wait?)

S. Kiranyaz and M. Gabbouj

**Abstract:** A novel multimedia retrieval technique, called progressive query (PQ) is presented. PQ is designed to bring an effective solution especially when querying large-scale multimedia databases. In addition, PQ produces intermediate query retrieval results during the execution of the query. The series of intermediate query results will finally converge to the full-scale search retrieval in a faster way and with no minimum system requirements. Experimental progressive query retrieval results show that intermediate retrieval results may be satisfactory and further query processing time may be avoided.

## 1 Introduction

It is a known fact that recent technological hardware and network improvements along with the daily usage of Internet have caused a rapid increase in the size of digital audio-visual information that is used, handled and stored via several applications. Besides several benefits and usages, such massive collections of information have brought storage and especially management problems. In order to overcome such problems several content-based indexing and retrieval techniques and applications have been developed such as the MUVIS system [1, 2], Photobook [3] VisualSEEK [4], Virage [5], and VideoQ [6]; some of which are designed to bring a framework structure for the multimedia items such as digital images and audio/video clips. The usual approach for indexing is to map database primitives into some high dimensional vector space, that is so-called feature domain. The feature domain may consist of several types of features (visual, aural, motion, etc.) as long as the database contains such items from which those particular features can be extracted. Among so many variations, careful selection of the feature sets allows capturing the semantics of the database items. Especially for large-scale multimedia databases the number of features extracted from the raw data is often kept large owing to the naïve expectation that it helps to capture the semantics better. Content-based similarity between two database items can then be assumed to correspond to the (dis-) similarity distance of their feature vectors. Henceforth, the retrieval of similar database items with respect to a given query (item) can be transformed into the problem of finding such database items that give feature vectors, which are close to the query feature vector. This is so-called query-by-example (QBE), which is one of the most common retrieval schemes. The basic QBE operation is called

Normal Query (NQ), and works as follows: using the available aural or visual features (or both) of the queried multimedia item (i.e. an image, a video clip, an audio clip, etc.) and all the database items, the similarity distances are calculated and then merged to obtain a unique similarity distance per database item. Ranking the items according to their similarity distances (to the queried item) over the entire database yields the query result.

Such an exhaustive search for QBE is costly and CPU intensive especially for large-scale multimedia databases since the number of similarity distance calculations is proportional to the database size. This fact brought a need for indexing techniques, which will organise the database structure in such a way that the query time and I/O access amount could be reduced. During the past three decades, several indexing techniques have been proposed. Many of these techniques are formed in a hierarchical tree structure that is used to cluster (or partition) the feature space. Both KD-tree [7] and R-tree [8] are the first examples of spatial access methods (SAMs). Afterwards several enhanced SAMs have been proposed. R\*-tree [9] provides a consistently better performance than the R-tree and R<sup>+</sup>-tree [10] by introducing a policy called 'forced reinsert'. Lin *et al.* proposed TV-tree [11], which uses so-called telescope vectors. Berchtold *et al.* [12] introduced X-tree, which is particularly designed for indexing higher dimensional data. X-tree avoids overlapping of region bounding boxes in the directory structure by using a new organisation of the directory and as a result of this X-tree outperforms both TV-tree and R\*-tree significantly. Still bounding rectangles can overlap in the higher dimensions. In order to prevent this, White and Jain proposed the SS-tree [13], an alternative to R-tree structure, which uses minimum bounding spheres instead of rectangles. Even though SS-tree outperforms R\*-tree, the overlapping in the high dimensions still occurs. Thereafter several other SAM variants are proposed such as SR-tree [14], S<sup>2</sup>-Tree [15], Hybrid-Tree [16], A-tree [17], IQ-tree [18], Pyramid Tree [19], NB-tree [20], etc. Especially for content-based indexing and retrieval in large-scale multimedia databases, SAMs have several drawbacks and significant weaknesses. By definition an SAM-based indexing scheme partitions and works over a single feature space. However, a multimedia database can have several feature types

(visual, aural, etc.), each of which might also have multiple feature subsets. Furthermore, SAMs assume that query operation time and complexity are only related to accessing a disc page (I/O access time) containing the feature vector. This is obviously not a trivial assumption for multimedia databases and consequently, no attempt in the design of SAMs has been done to reduce the similarity distance computations (CPU time). In order to provide a more general approach to similarity indexing for multimedia databases, several efficient metric access methods (MAMs) have been proposed. The generality of MAMs comes from the fact that any MAM employs the indexing process by assuming only the availability of a similarity distance function, which satisfies three trivial rules: symmetry, non-negativity and triangular inequality. Therefore, a multimedia database might have several feature types along with various numbers of feature sub-sets all of which are in different multi-dimensional feature spaces. As long as a similarity distance function that is usually treated as a 'black box' by the underlying MAM, exists the database can be indexed by any MAM. Several MAMs have been proposed so far. Yianilos [21] presented vp-tree that is based on partitioning the feature vectors (data points) into two groups according to their similarity distances with respect to a reference point, so called vantage point. Bozkaya and Ozsoyoglu [22] proposed an extension of vp-tree, so-called mvp-tree (multiple vantage point), which basically assigns  $m$  vantage points for a node with a fan out of  $m^2$ . They reported 20 to 80% reduction in the similarity distance computation time compared to vp-trees. Brin [23] introduced the geometric near-neighbour access tree (GNAT) indexing structure, which chooses  $k$  number of split points at the top level and each of the remaining feature vectors are associated with the closest split points. GNAT is then built recursively and the parameter  $k$  value is chosen to be a different value for each feature set depending on its cardinality. The aforementioned MAMs have several shortcomings. Contrary to SAMs, these metric trees are designed only to reduce the number of similarity distance computations, paying no attention to I/O costs (disc page accesses). They are also static methods in the sense that the tree structure is built once and new insertions are not supported. Furthermore, all of them build the indexing structure from top to bottom and hence the resulting tree is not guaranteed to be balanced. Ciaccia *et al.* [24] proposed M-tree to overcome such problems. M-tree is a balanced and dynamic tree, which is built from bottom to top, creating a new root level only when necessary. The node size,  $M$ , is a fixed number and therefore, the tree height depends on  $M$  and the database size. Its performance optimisation concerns both CPU computational time for similarity distances and I/O costs for disc page accesses for feature vectors of the database items. Recently Traina *et al.* [25] proposed Slim-tree, an enhanced variant of M-tree, which is designed for improving the performance by minimising the overlaps between nodes. They introduced two factors, 'fat-factor' and 'bloat-factor', to measure the degree of overlap and proposed the usage of minimum spanning tree (MST) for splitting the node. Another slightly enhanced M-tree structure, so-called  $M^+$ -tree, can be found in [26].

Along with the indexing techniques addressed so far, certain query techniques are needed to speed up a QBE process. The most common query techniques developed for the aforementioned indexing techniques are as follows:

- Range queries: Given a query object,  $Q$ , and a maximum similarity distance range,  $\varepsilon$  and the similarity distance

function  $SD$ , the range query selects all indexed database items,  $Q_i$ , such that  $SD(Q, Q_i) < \varepsilon$ .

- $kNN$  queries: Given a query object,  $Q$ , and an integer number  $k > 0$ ,  $kNN$  query selects the  $k$  database items, which have the shortest similarity distance from  $Q$ .

Unfortunately, both query techniques may not provide an efficient retrieval scheme from the user's point of view owing to their parameter dependency. For instance, range queries require a distance parameter,  $\varepsilon$ , where the user may not be able to provide such a number prior to a query process since it is not obvious to find out a suitable range value if the database contains various types of features and feature subsets. Similarly, parameter  $k$  in a  $kNN$  query may be hard to determine since it can be too small in case the database may provide many more similar (relevant) items than required, or too big if the number of similar objects is only a small fraction of the required number  $k$ , which means unnecessary CPU time has been wasted for that query process. Both query techniques often require several trials to converge to a successful retrieval result and this alone might remove the speed benefit of the underlying indexing scheme, if there is any.

As mentioned before, the other alternative is the so-called Normal Query (NQ), which makes a sequential (exhaustive) search owing to lack of an indexing scheme. NQ for QBE is costly and CPU intensive especially for large-scale multimedia databases; however, it yields such a final and decisive result that no further trials are needed. Still, all QBE alternatives have some common drawbacks. First of all, the user has to wait until all (or some) of the similarity distances are calculated and the searched database items are ranked accordingly. Naturally, this might take a significant time if the database size (or  $k, \varepsilon$ ) is large and the database contains a rich set of aural and visual features, which might further reduce the efficiency on the indexing process. Furthermore, any abrupt stopping (i.e. manual stop by the user) during the query process will cause total loss of retrieval information and essentially nothing can be saved out of the query operation so far performed. In order to speed up the query process, it is a common application design procedure to hold all features of database items into the system memory first and then perform the calculations. Therefore, the growth in the size of the database and the set of features will not only (proportionally) increase the query time (the time needed for completing a query) but it might also increase the minimum system memory requirements such as memory capacity and CPU power.

In order to eliminate such drawbacks and provide a faster query scheme, we have developed a novel retrieval scheme, the Progressive Query (PQ), which is implemented under the MUVIS system to provide a basis for multimedia retrieval and to test the performance of the technique. PQ is a retrieval (via QBE) technique, which can be performed over the databases with or without the presence of an indexing structure. Therefore, it can be an alternative to NQ where both produce (converge to) the same result at the end. When the database has an indexing structure, PQ can replace  $kNN$  and range queries whenever a query path over which PQ proceeds, can be formed. As its name implies, PQ provides intermediate query results during the query process. The user may browse these results and may stop the ongoing query in case the results obtained so far are satisfactory and hence no further time should be wasted unnecessarily. As expected, PQ and NQ will converge to the same (final) retrieval results at the end. Furthermore, PQ may perform the overall query process faster (within a shorter total query time) than NQ. Since PQ provides

a series of intermediate results, each of which obtained from a (smaller) sub-set within the database, the chance of retrieving relevant database items that would not be retrieved otherwise via NQ, might be increased. Approvingly some experimental results show that it is quite probable to achieve even better retrieval performance within an intermediate sub-query than the final query state.

It is a known fact that significant performance improvements of content-based multimedia retrieval systems can be achieved by using a technique known as relevance feedback [27, 28], which allows the user to rate (intermediary) retrieval results. This helps to tune the ranking and retrieval parameters and hence yields a better retrieval result at the end. Traditional query techniques so far addressed (NQ,  $kNN$  and range queries) may allow such a feedback only after the query is completed. Since PQ provides the user with intermediate results during the query process, relevance feedback may be applied already to these intermediate results, yielding possibly faster satisfactory retrieval results from the user's point of view.

## 2 Progressive query overview

The principal idea behind the design of PQ is to partition the database items into some sub-sets within which individual (sub-) queries can be performed. Therefore, a sub-query is a fractional query process that is performed over any sub-set of database items. Once a sub-query is completed over a particular sub-set, the incremental retrieval results (belonging only to that sub-set) should be fused (merged) with the last overall retrieval result to obtain a new overall retrieval result, which belongs to the items where PQ operation so far covers from the beginning of the operation. Note that this is a continuous operation, which proceeds incrementally, sub-set by sub-set, by covering more and more groups of items within the database. Each time a new sub-query operation is completed, PQ updates the retrieval results to the user. Since the previous (overall) query results are used to obtain the next (overall) retrieval result via fusion, the time consuming query operation is only performed over the (next) partitioned group of items instead of all the items where PQ covered so far.

The order of the database items processed is a matter for the indexing structure of the database. If the database is not indexed at all, simply a sequential or random order can be chosen. In case the database has an indexing structure, a query path can be formed in order to retrieve the most relevant items at the beginning during a PQ operation.

Since there are various indexing schemes addressed in the previous Section, for the sake of simplicity, we shall first explain the basics of PQ for a database with no indexing structure.

Another important factor is to determine the size of each sub-set (i.e. the number of items within a sub-set where sub-query operation is performed) that is most convenient from the user's point of view. A straightforward solution is to let the user fix the sub-set size (say e.g. 25). This would mean that the user wants updates every time 25 items are covered during the ongoing PQ operation. However, this also brings the problem of uncertainty because the user cannot know how much time a sub-query will take beforehand since the sub-query time will vary owing to factors such as the amount of features present in the database and the speed of the computer where it is running, etc. Therefore the PQ retrieval updates might be too fast or too slow for the user. To avoid such uncertainties, the proposed PQ scheme is designed over periodic sub-queries as shown in Fig. 1 with a user defined period value ( $t = t_p$ ). The period (time) is an obviously more natural choice since the user can eventually expect the retrieval results will be updated every  $t_p$  seconds no matter what database is involved or what computer is used. Without loss of generality, in databases without an indexing structure, PQ is designed to perform sub-set partitioning sequentially with a forward direction (i.e. starting from the first item to the last one).

Next, we shall first present the formation of periodic sub-queries and the mechanism needed to partition the database into such sub-sets that each progressive sub-query (PSQ) retrieval result can be updated within a close neighbourhood of a user-defined period ( $t_p$ ). Sub-query fusion operation will then be explained in Section 2.2. Finally, Section 2.3 is devoted specifically to PQ for databases with an (hypothetical) indexing structure.

### 2.1 Periodic sub-query formation

To achieve periodic sub-queries, we need to define some additional sub-query compositions.

**2.1.1 Atomic sub-query:** This is the smallest sub-set size on which a sub-query is performed. Here we assume that atomic sub-query time is not significant compared to periodic sub-query time. Atomic sub-queries are the only sub-query types that have a fixed sub-set size ( $S_{ASQ}$ ). They are only used during a first periodic query and they are used

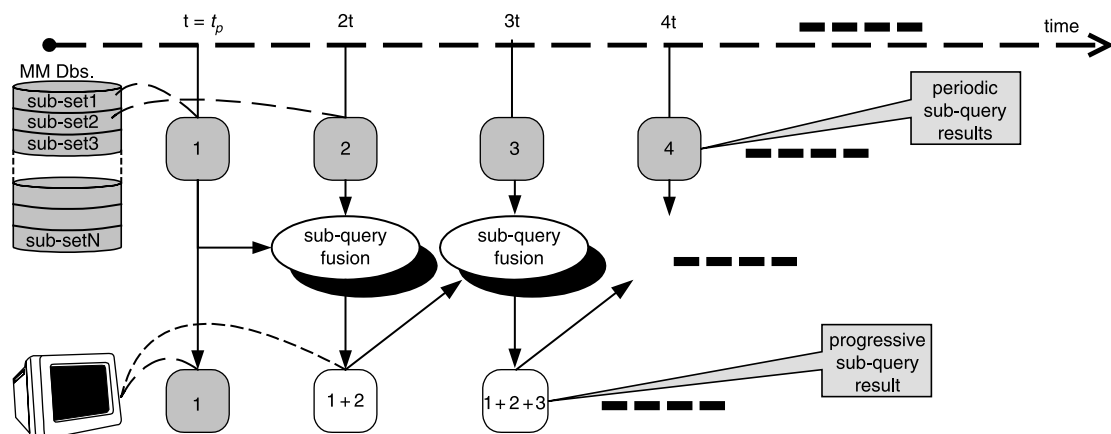


Fig. 1 Progressive query overview

in order to provide an initial sub-query per item time ( $t_r^0$ ), that is the time spent for the retrieval of a single database item, formulated as follows:

$$t_r^0 = \frac{t_{ASQ}}{N_{ASQ}} \text{ if } N_{ASQ} > 0 \quad (1)$$

where  $t_{ASQ}$  is the total time spent for atomic sub-query and  $N_{ASQ}$  is the number of database items that are involved (used) in the atomic sub-query operation. Without an indexing structure, note that  $0 \leq N_{ASQ} \leq S_{ASQ}$ ; since the initial database items might not belong to the ongoing query type. For example in a multimedia database, there might be video-only clips and audio-only clips (and clips with both media types). So for a visual query, those audio-only clips will be discarded totally and if the initial atomic query sub-set covers such audio-only clips then naturally  $N_{ASQ} \leq S_{ASQ}$ . In case  $N_{ASQ} = 0$ , one or more atomic sub-queries have to be performed until we get a valid  $t_r^0$  value (i.e.  $N_{ASQ} > 0$ ).

**2.1.2 Fractional sub-query:** This can be any sub-query performed over a sub-set whose size is smaller or equal to the sub-set size of the periodic sub-query. That is, a fractional sub-query time might be less than or equal to a periodic sub-query time.

As explained previously, periodic sub-queries are periodic over time and a mechanism is needed to ensure this periodicity. This mechanism works over atomic and fractional sub-queries; it performs fusion operation over as many atomic and fractional sub-queries as necessary. First, it starts with an atomic sub-query to obtain a valid (initial) sub-query per item time,  $t_r^0$ , and it keeps going with atomic queries until a valid  $t_r^0$  value is obtained. Once it is obtained, then one or more fractional sub-queries will be performed to complete the first periodic sub-query. The size of the fractional query ( $N_{FSQ}$ ) can then be estimated as:

$$N_{FSQ}^0 \cong \frac{t_p^0}{t_r^0} \quad (2)$$

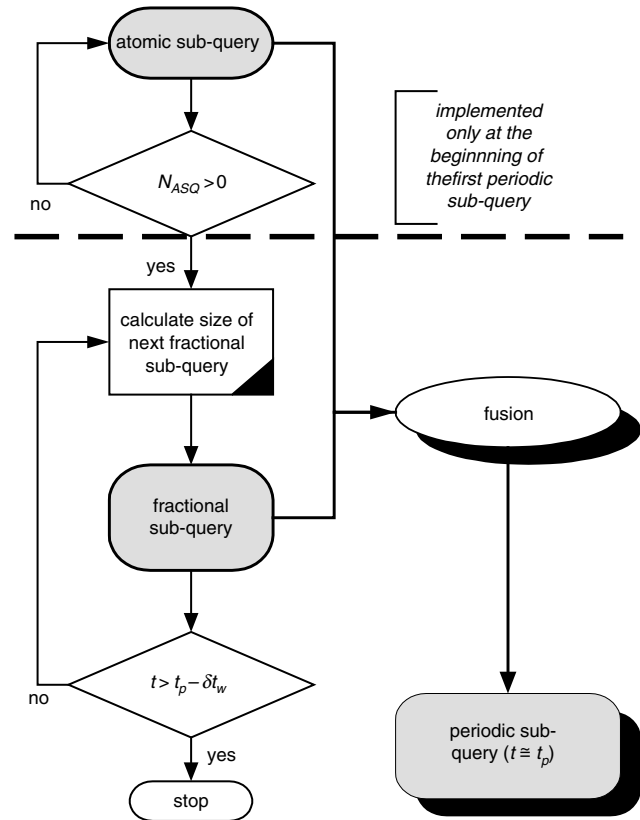
where  $t_p^0 = t_p - t_\Sigma^a$  is the time left for completing the first periodic sub-query and  $t_\Sigma^a$  the total time spent for all atomic queries performed so far. Afterwards, the fractional sub-query can be performed within a sub-set of  $N_{FSQ}^0$  items. Once the fractional sub-query is completed, the total time ( $t_\Sigma$ ) so far spent from the beginning of the operation till now is compared with the required time period of the  $q$ th (so far  $q = 0$ ) periodic query,  $t_p^q$ , where  $q$  is the periodic sub-query index. If the  $t_\Sigma$  value is not within a close neighborhood (i.e.  $\delta t_w < 0.5$  s) of  $t_p^q$  (i.e.  $t_\Sigma < t_p^q - \delta t_w$ ) then the operation continues with a new fractional sub-query until the condition is met. For the new fractional sub-query and for all the latter fractional sub-query operations  $t_r$  value is re-estimated (updated) from the former operations such as:

$$N_{FSQ}^i = \frac{t_p^q - t_\Sigma}{t_r^i} \quad \text{--- } t_\Sigma < t_p^q - \delta t_w \text{ ---}$$

$$t_r^{i+1} = \frac{t_\Sigma}{\sum_{i \in FSQ} N_{FSQ}^i} \text{ if } \sum_{i \in FSQ} N_{FSQ}^i > 0 \quad (3)$$

Once one or more fractional queries form the  $q$ th periodic query, owing to offset that has occurred from the period of PQ, next periodic sub-query ( $q + 1$ st) is formed with an updated (offset removed) period value:

$$t_p^{q+1} = t_p + (t_p^q - t_\Sigma) \quad (4)$$

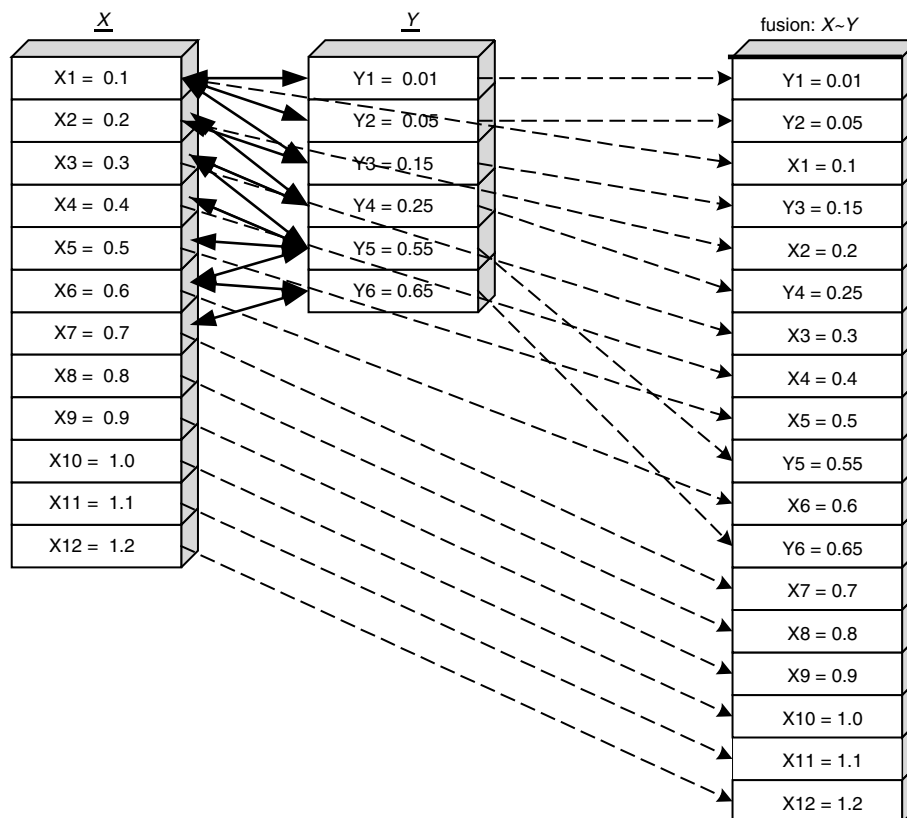


**Fig. 2** Formation of a periodic sub-query

where  $t_p$  is the required period and  $(t_p^q - t_\Sigma)$  is the offset time, which is the time difference between the required period time for  $q$ th periodic sub-query and the total (actual) time spent so far. The flowchart of the formation of a periodic sub-query is shown in Fig. 2.

## 2.2 Sub-query fusion operation

The overall PQ operation is carried out over progressive sub-queries (PSQs). In principal, it can be stated that PQ is a (periodic) series of PSQ results. As shown in Fig. 1, a new PSQ retrieval is formed each time a periodic sub-query is completed and then it is fused with the previous PSQ retrieval result. Once a PSQ is realised, the results are shown to the user and saved during the lifetime of the ongoing PQ so that the user can access them at any time. The user is shown updated retrieval results each time a new PSQ is completed. The first PSQ is the first periodic sub-query performed. The fusion operation is a process of fusing two sorted sub-query results to achieve one (fused) sub-query result. Since both of the sub-query results are already sorted with respect to the similarity distances, simply comparing the consecutive items in each of the sub-query lists can perform the fusion operation. Let  $n_1$  and  $n_2$  be the number of items in the first and second sub-set, respectively. Since there are  $n_1 + n_2$  items to be inserted into the fused list one at a time, the fusion operation can take a maximum  $n_1 + n_2$  comparisons. This (worst) case occurs whenever the items from both lists are distributed evenly with respect to each other. Alternatively if the maximum valued item (i.e. the last item) in the smaller list is less than the minimum valued item (i.e. the first item) in the bigger list, the number of comparisons will not exceed the number of items in the smaller list because once all of the items in it are compared with the (smallest) first item in the bigger list and henceforth inserted into the fused list, there will not be any more comparisons needed. Note that this is the direct



**Fig. 3** A sample fusion operation between sub-sets  $X$  and  $Y$

consequence of the fact that the both lists are sorted (from minimum to maximum) beforehand and one of them is now fully depleted. Therefore, the fusion operation will take minimum  $Min(n_1, n_2)$  comparisons respectively. A sample fusion operation is illustrated in Fig. 3. Note that the subsets  $X$  and  $Y$  contain 12 and 6 items, respectively, and the fusion operation performs only 12 comparisons.

Since the fusion operation is nothing but merging two arbitrarily sized sub-sets (retrieval results), it can be applied during each phase of a PQ operation. For instance, the atomic sub-queries are fused with fractional sub-queries and several fractional sub-queries are fused to obtain a periodic sub-query. Fusing the periodic sub-query with the previous PSQ retrieval produces a new PSQ retrieval, covering a larger part of the database. If the user does not stop the ongoing PQ operation, it will eventually cover the entire database at the end and therefore, it generates the overall retrieval result of the queried item. In this case PQ generates the exact same retrieval result as NQ since both of them perform the same operation, i.e. searching a queried item through the entire database and ranking the database items accordingly.

### 2.3 PQ in indexed databases

For the databases without an indexing structure, PQ can be used conveniently as an alternative query scheme to the traditional query type, NQ. As a retrieval process, PQ can also be performed over indexed databases as long as a query path can be formed over the clusters (partitions) of the underlying indexing structure. Obviously, a query path is nothing but a special sequence of the database items. When the database lacks an indexing structure, it can be formed in any convenient way such as sequentially (starting from the 1st item towards the last one, or vice versa) or randomly. Otherwise, the most advantageous way to perform PQ is to use the indexing information so that the most relevant items

can be retrieved in earlier PSQ steps. Since the technical details of various available indexing schemes are beyond the scope of this paper, we only show the hypothetical formation of the query path and run-time evaluation of PQ over this path.

As briefly mentioned earlier, the primary objective of indexing structures is to partition the feature domain into such (tree-based) clusters that CPU time and I/O accesses are shortened via pruning of the redundant tree nodes. Figure 4 shows a hypothetical clustering scheme and the formation of the query path over which PQ will proceed during its run-time. The illustration shows 4 clusters (partitions or nodes), which contain a certain number of items (features) and the query path is formed according to the relative (similarity) distance to the queried item and its parent cluster. Therefore, PQ will give the priority to cluster A (the host), then B (the closest), C, D, etc. Note that the query path might differ from the final retrieval result depending on the accuracy of the indexing scheme. For instance, query path gives priority to item B2 on the search with respect to item C4 but item C4 may have more similarity (relevancy) with respect to the queried item A2. When the retrieval results are formed it will eventually be ranked higher and presented earlier to the user by PQ. Even though PQ corrects this misleading result owing to the erroneous indexing (note that in this case item C4 should have belonged to cluster B, not C), as a possible consequence of this, the retrieval of C4 might be delayed to the next periodic PSQ retrieval.

At this point, one can implement two different approaches: the overall query path can be formed immediately after the query is initiated and then the PQ evolves over it with its natural supplies of periodic retrievals. This approach is only recommended for small and medium sized databases where the complete query path formation takes insignificant time. Otherwise, the query path should be

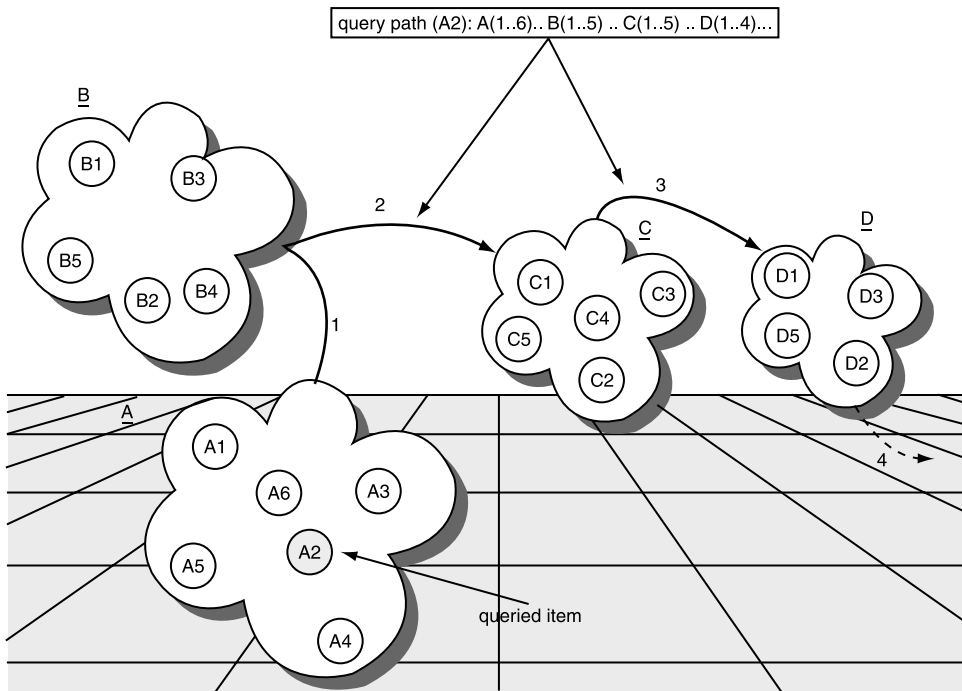


Fig. 4 Query path formation in a hypothetical indexing structure

dynamically (incrementally) formed along with the PQ run-time process and the time spent for it should be taken into account during the adaptive calculation of period given in (4). In this case, the adaptive period calculation for the  $q + 1$ st periodic sub-query period should be reformulated as follows:

$$t_p^{q+1} = t_p + \left( t_p^q - t_\Sigma - t_{QP}^q \right) \quad (5)$$

where  $t_{QP}^q$  is the time spent for forming the query path during the formation of  $q$ th periodic sub-query.

As a result PQ in indexed databases makes more sense than to be strictly dependant on an unknown parameter such as  $k$  as in  $kNN$  query or  $\epsilon$  in range query, which might cause a deficiency in the retrieval performance such as the casual need for doing multiple queries to come with a satisfactory result at the end. Alternatively there exists a certain degree of similarity (or analogy) between PQ and those conventional query techniques. For instance each PSQ retrieval can be seen as a particular  $kNN$  (or range) query retrieval with only one difference: the parameter  $k$  (or  $\epsilon$ ) is not fixed beforehand, rather dynamically changing (growing) over time along with the lifetime of PQ and the user has the opportunity to fix it (stop PQ) whenever satisfactory retrievals are obtained.

### 3 Experimental results

To present the experimental conditions, the multimedia databases used and especially the test-bed platform used for experiments, Section 3.1 briefly introduces MUVIS and particularly the *MBrowser* application under which PQ is primarily developed and tested. Later in Section 3.2 we begin the evaluation of PQ with respect to NQ in terms of speed, memory, accessibility and (better) relevancy in the databases with no indexing structure and hence without interference of any indexing algorithm. We shall also present the evaluation of PQ with respect to the variations of the query parameters such as PQ period ( $t_p$ ), query type, etc. and show some sample visual and aural PQ retrievals in image, video and audio databases.

#### 3.1 PQ within MUVIS

MUVIS framework aims to bring a unified and global approach to indexing, browsing and querying of various multimedia types such as audio/video clips and still images. MUVIS basically provides tools for real-time audio and video capturing, encoding by several widely used codecs such as MPEG-4, H.263+, MP3 and AAC. It supports several digital image types including JPEG-2000. Furthermore, it provides a well-defined interface to integrate third party feature extraction algorithms into the framework.

As shown in Fig. 5, MUVIS framework is based upon three applications, each of which has different responsibilities and facilities. *AVDatabase* is mainly responsible for real-time audio/video database creation with which audio/video clips are captured, (possibly) encoded and recorded in real-time from any peripheral audio and video devices connected to a computer. *DbsEditor* performs the indexing of the multimedia databases and therefore, off-line feature extraction process over the multimedia collections is its main task. *MBrowser* is the primary media browser and retrieval application into which the PQ technique is integrated as the primary retrieval (QBE) scheme. NQ is the alternative query scheme within *MBrowser*. Both PQ and NQ can be used for ranking the multimedia primitives with respect to their similarity to a queried media item (an audio/video clip, a video frame or an image). Owing to their unknown duration, which might cause impractical indexing times for an on-line query process, to query an (external) audio/video clip, it should first be appended (off-line operation) to a MUVIS database upon which the query will be performed. There is no such necessity for images; any digital image (inclusive or exclusive to the active database) can be queried within the active database. The similarity distances will be calculated by the particular functions, each of which is implemented in the corresponding visual/aural feature extraction (*FeX* or *AFeX*) modules. More detailed information about MUVIS can be found in [1] and [2].

As shown in Fig. 6, the *MBrowser* GUI is designed to support all the functionalities that PQ provides. Once a MUVIS database is loaded into *MBrowser*, the user

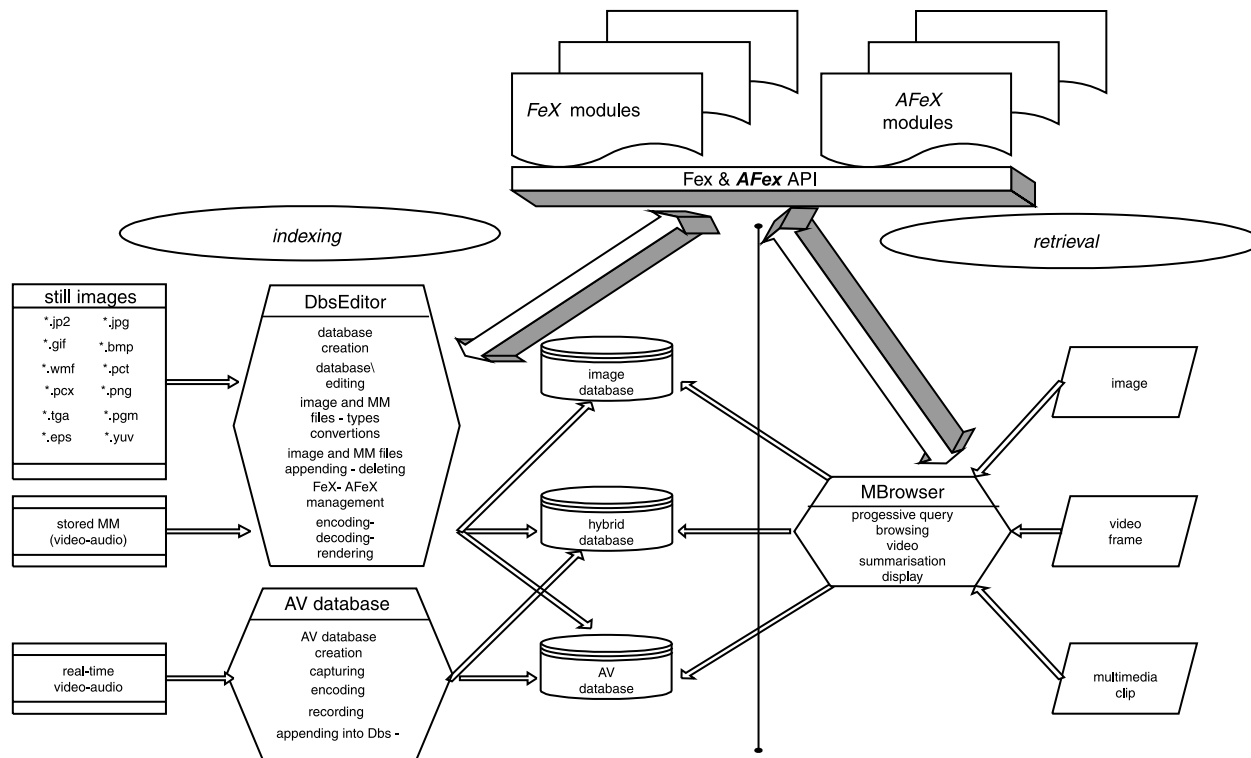


Fig. 5 Generic overview of MUVIS framework

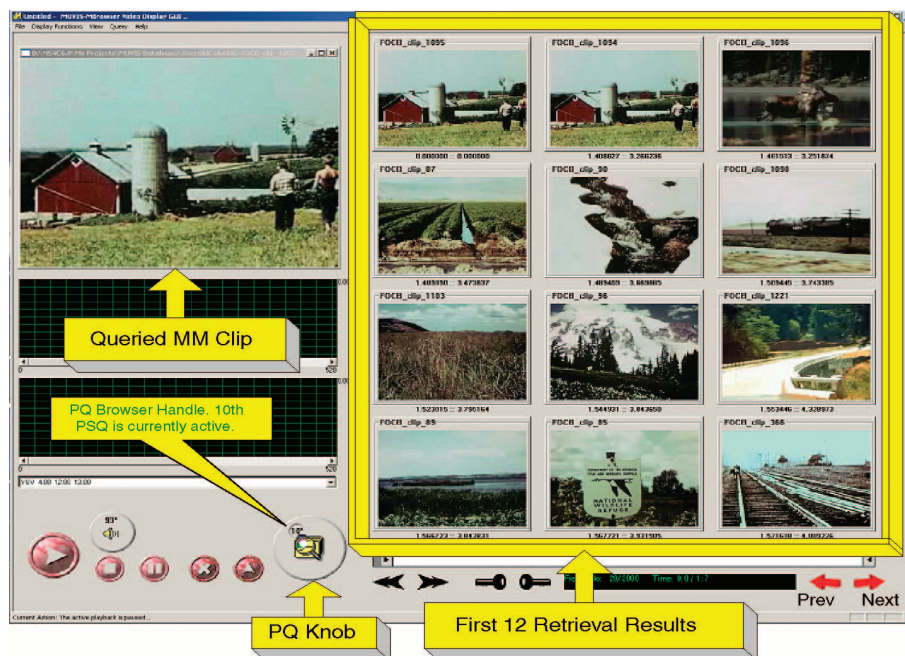


Fig. 6 MBrowser GUI showing a PQ operation where 10th PSQ is currently active (or set manually)

can browse among the database items, choose any item and then initiate a query. The basic query parameters such as query type (PQ or NQ), query genre (aural or visual), PQ update period (time), the (visual and aural) set of features and their individual parameters (i.e. feature weights), etc. can be set prior to a query operation. When a (sub-) query is completed the retrieval results are then presented to the user page by page. Each page renders 12 ranked results in the descending order (from left to right and from top to bottom) and the user can browse the pages back and forth using the *Next* and *Prev* buttons on the bottom right of the Figure (the first page with 12-best retrieval results is

shown on the right of Fig. 6). If NQ is chosen, then the user has to wait till the whole process is completed but if PQ is chosen then the retrieval results will be updated periodically (with the user-defined period value) each time a new PSQ is accomplished. The current PSQ number (10) is shown on the *PQ Browser Handle* and this handle can also be used to browse manually among the retrieved PSQ results during (or after) an ongoing PQ operation. In the snapshot shown in Fig. 6, a video clip is chosen within a MUVIS (video) database and visual PQ is performed. Currently the 1st page (12-best results) of the 10th PSQ retrieval results is shown on the GUI window of MBrowser.

In the experiments performed in this section, we used 4 different MUVIS databases:

- 1) *Open Video* database: This database contains 1500 video clips, each of which is downloaded from ‘The Open Video Project’ web site [29]. The clips are quite old (from the 1960s) but contain colour video with sound. The total duration of the database is around 46 hours.
- 2) *Real World* audio/video database: There are 800 audio-only clips and video clips in the database with a total duration of over 36 hours. They are captured from several TV channels and the content is distributed among news, advertisements, talk shows, cartoons, etc.
- 3) *Sports* hybrid database: There are 200 video clips mainly carrying sports content such as football, tennis and formula-1. There are also 495 images (in GIF and JPEG formats) showing instances from football matches and other sports tournaments.
- 4) *Shape* image database: There are 1500 black and white (binary) images that represent mainly the shapes of different objects such as animals, cars, accessories, geometric objects, etc.

All experiments are carried out on a Pentium-4 3.06 GHz computer with 2048 MB memory. In order to have unbiased comparisons between PQ and NQ, the experiments are performed using the same queried multimedia item with the same instance of *MBrowser* application. The evaluations of the retrieval results by QBE are performed subjectively using ground-truth method, i.e. a group of people evaluates the query results of a certain set of retrieval experiments, upon which all the group members totally agreed about the query retrieval performance. Among these a certain set of examples were chosen and presented in this article for visual verification.

### 3.2 PQ against NQ

As explained earlier, PQ and NQ eventually converge to the same retrieval result at the end. Also in the abovementioned scenarios they are both designed to perform an exhaustive search over the entire database within MUVIS. However PQ have several advantages over NQ as detailed in the following Sections.

**3.2.1 System memory requirement:** The memory requirement is proportional to the database size and the number of features present in a NQ operation. Owing to the partitioning of the database into sub-sets, PQ will reduce the memory requirement by the number of PSQ operations performed. After each periodic sub-query operation, the memory used for feature vectors in that sub-set is no longer needed and can be used for the next periodic sub-query. Figure 7 illustrates the memory usage of a retrieval example that is shown later in Fig. 9 by a PQ first

and a NQ afterwards. As clearly seen in this experimentation, PQ only requires a fractional memory per sub-query operation and it can free it at the end of each PSQ retrieval, whereas NQ needs to keep a big amount of memory for allocations of features used and similarity distances calculated afterwards, till to the end of the sorting (ranking) and rendering (on screen) operations that are naturally reached only at the end.

We also observe that especially in very large-scale databases containing a rich set of features, NQ can exceed the system memory. Two possible outcomes may eventually occur as a result. The operating system may handle it by using virtual memory (i.e. disc) if the excessive memory requirement is not too high. In this case, the operational speed for a NQ operation will be degraded drastically and eventually PQ can outperform NQ several times with respect to overall retrieval time. The other possibility is that NQ operation cannot be completed at all since the system is not capable of providing the excessive memory required by NQ and in this case PQ is the only feasible query operation.

**3.2.2 Earlier/better relevant results:** A better retrieval result occurs when more relevant items are ranked higher in the retrieval result. An earlier (and equal) result occurs when the same end result is achieved in an earlier time instance. Along with the ongoing process PQ allows intermediate query results (PSQ steps), which might sometimes show equal or ‘even better’ performance than the final (overall) retrieval result as some typical examples given in Fig. 8 and Fig. 9.

In Fig. 8, an image retrieval example within *Shape* database via PQ using canny edge histogram feature is shown. We use  $t_p = 0.2$  s and PQ operation is completed in three PSQ series (i.e. PQ #1, #2 and #3). This is one particular example in which an intermediate PSQ retrieval yields a better performance than the final PQ retrieval (that is the same as the retrieval result of NQ). In this example, the first 12-best retrieval results in PQ #1 are obviously better (more relevant in terms of shape similarity to the queried shape, i.e. ‘mobile phone’) than the ones in PQ #3 (NQ).

In Fig. 9, a video retrieval example within *Real World* database via aural PQ using mel-frequency cepstral coefficients (MFCC) as the audio features is shown. We use  $t_p = 5$  s and PQ operation is completed in 12 PSQ series but only 3 PSQ retrievals (i.e. PQ #1, #6 and #12) are shown. Note that PQ #6 and the latter retrieval results till PQ #12 are identical, which means that PQ operation produces the final retrieval result (which NQ would produce) in an earlier (intermediate) PSQ retrieval.

Such ‘earlier and even better’ retrieval results can be verified owing to the fact that searching an item in a smaller data set usually yields better (detection or retrieval) performance than searching in a larger set. This is obviously

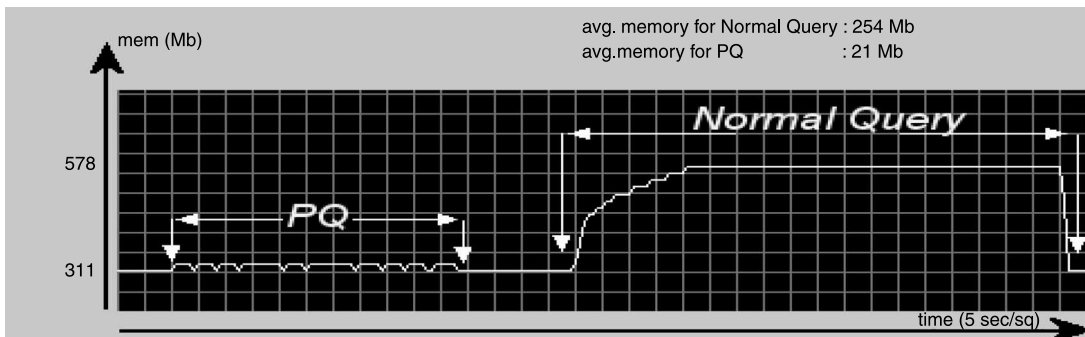
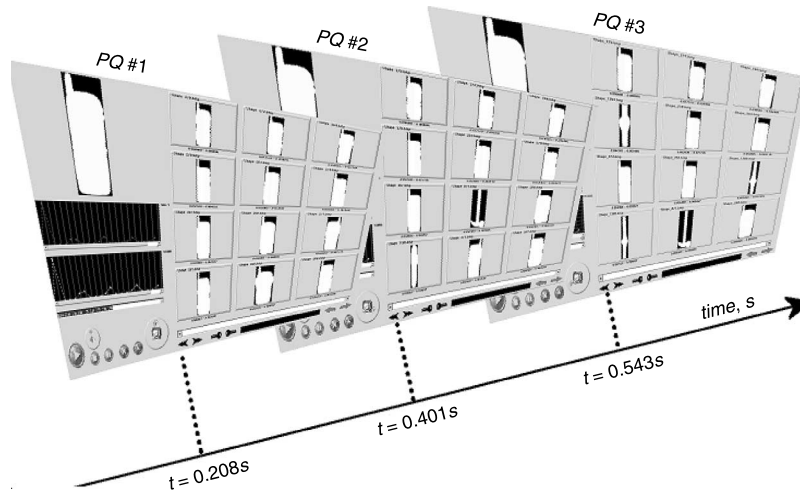
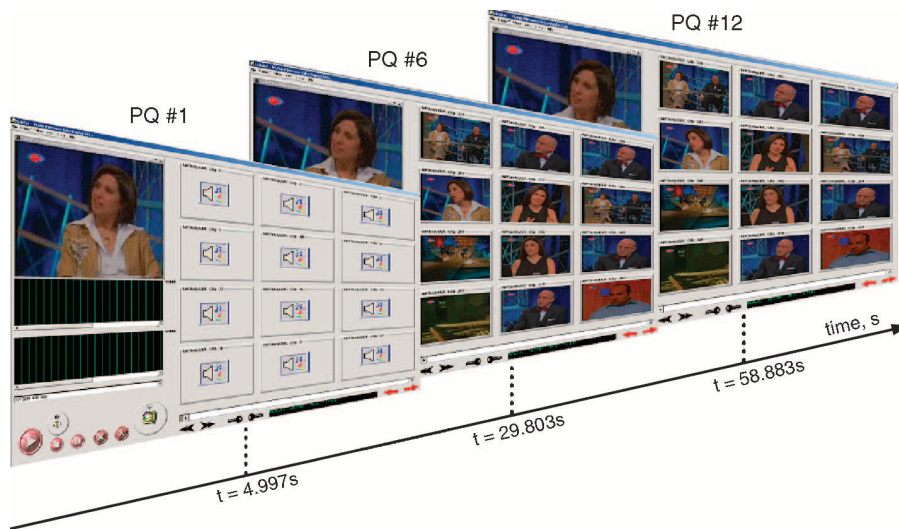


Fig. 7 Memory usage for PQ and NQ



**Fig. 8** PQ retrievals of a query image (left-top) within 3 PSQs ( $t_p = 0.2$  s)



**Fig. 9** Aural PQ retrievals of a video clip (left-top) in 12 PSQs (Only 1st, 6th and 12th are shown with  $t_p = 5$  s)

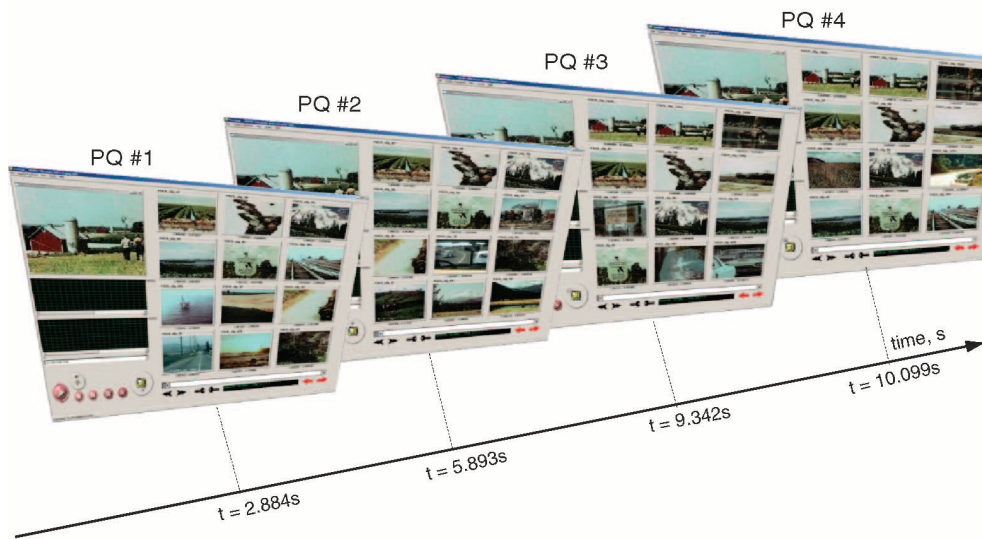
an advantage for PQ since it proceeds within sub-queries performed in (smaller) sub-sets whereas NQ always has to proceed through the entire database. Furthermore, for the databases that are not indexed such as in the examples given, this basically means that the order of the relevant items coincides with the progress of the ongoing PQ operation in the earlier PSQ steps. When the database has a solid indexing structure and a query path can be formed according to the relevancy of the queried item, the user eventually gets relevant retrieval results in a fraction of the time that is needed for a typical NQ operation.

**3.2.3 Query accessibility:** This is the major advantage that PQ provides. Stopping an ongoing query operation is an important capability. As shown in Fig. 6, by pressing the PQ Knob during an ongoing PQ operation, the user can stop it any time (i.e. when the results are so far satisfactory). Of course, NQ can also be stopped but no retrieval result is available afterwards since the operations such as similarity distance calculations or sorting are likely not to be completed.

Another important accessibility option that PQ offers is so-called PSQ browsing. When stopped abruptly or completed at the end, the user can still browse among PSQ retrievals and visit any retrieval page of that particular PSQ since the retrieval results will be alive unless a new PQ is initiated or the application is terminated. This is obviously

a significant requirement especially when better results are obtained in an earlier PSQ step than the later ones as mentioned before. Alternatively, this might still be a desirable option even if the earlier PSQ results are not better but comparable as much as the later ones. This could be relevant to the user. One particular example is shown in Fig. 10, that is, a video retrieval example from the *Open Video* database via visual PQ using several colour (YUV, HSV, etc.), texture (GLCM [30]) and shape (canny edge histogram) features. We use  $t_p = 3$  s and PQ operation is completed in 4 PSQ series. Note that a retrieval performance criteria can be difficult to accomplish among these PSQ retrievals since their relevancy to the queried item is subjective.

The most important accessibility advantage that PQ can provide is that it can further improve the efficiency of any relevance feedback mechanism in certain ways. An ordinary relevance feedback technique works as follows: the user initiates a query and only after the query is completed, the user gives some feedback to the system about the retrieval results according to their relevancy (and/or irrelevancy) with respect to the queried item. Afterwards, a new query is initiated to get better retrieval results and this might be repeated several times until satisfactory results are obtained. This is an especially time consuming process since at each iteration the user has to wait until the query operation is completed. Owing to the enhanced accessibility options that



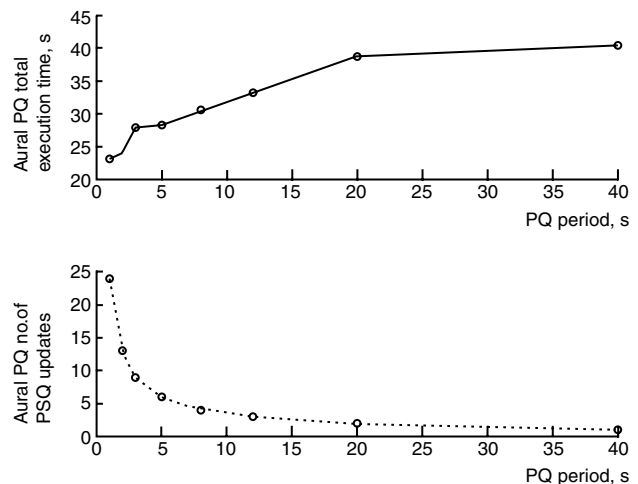
**Fig. 10** Visual PQ retrievals of a video clip (left-top) in 4 PSQs ( $t_p = 3$  s)

PQ provides, significant improvements can be achieved for the user with the following scenarios. First the user can employ relevant (and irrelevant) feedbacks during the query process and the incoming progressive retrievals can thus be tuned progressively. This means that during an ongoing query process the user can employ one or more relevant feedbacks anytime (within the life-time of PQ). Another alternative is that, the user can stop an ongoing PQ and then employs the relevant feedbacks with respect to the (intermediate) retrievals via PSQ browsing and re-initiate a new (fine-tuned) PQ. Basically any relevance feedback technique can be applied along with PQ since in both scenarios PQ only provides the necessary basis for the (user) accessibility to employ the relevance feedback but otherwise stay independent from the internal structure of any individual technique employed.

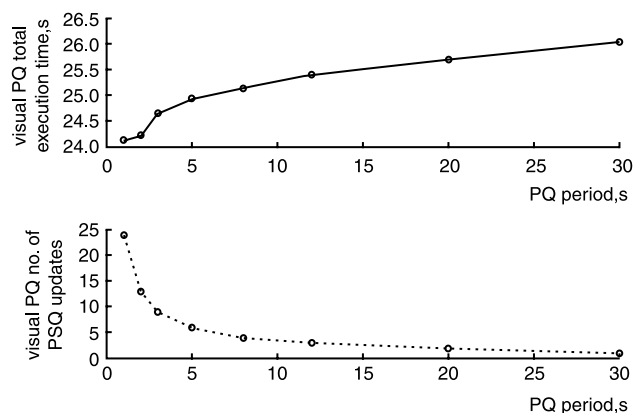
**3.2.4 Overall retrieval time (query speed):** The overall query time is the time elapsed from the beginning of the query to the end of the operation. For NQ, this is obviously the total time from the moment the user initiates the query until the results are ranked and displayed on the screen. However, for PQ, since the retrieval is a continuous process with PSQ series, the overall retrieval means that PQ proceeds over the entire database until the process terminates. As mentioned earlier, at this point both PQ and NQ will generate identical retrieval results for a particular queried item. The retrieval process includes the execution times for loading the database features from the disc to the computer memory, calculating the similarity measure between the query item and the database items and finally ranking the results. Both NQ (single stage) and PQ (multiple stages) would take the same amount of time to execute the first two tasks; while, PQ will execute the third task (i.e. ranking the results) faster than NQ since, at each PSQ, fewer items are compared. In the limit, as the number of data partitions approaches the total number of items in the database, say  $N$ , ranking would require order  $N$  operations.

In order to verify this expectation, several audio PQ retrieval experiments in *Real World* database have been performed with different  $t_p$  values. To get an unbiased measure, the experiments for each  $t_p$  value are repeated 5 times and the median from 5 overall retrieval times is taken into account. PQ total execution time (overall retrieval time) and the number of PSQ updates are plotted in Fig. 11.

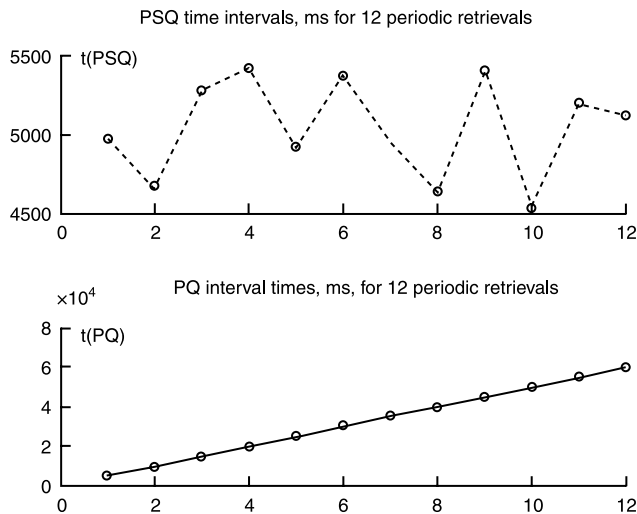
The same experiment is repeated for visual PQ operation and the result is shown Fig. 12. Note that if PQ is completed with only one PSQ (corresponding to the longest PQ period in Figs. 11 and 12), then it basically performs a NQ operation and therefore NQ retrieval time can also be examined in both figures.



**Fig. 11** Aural PQ overall retrieval time and PSQ number against PQ period



**Fig. 12** Visual PQ overall retrieval time and PSQ number against PQ period



**Fig. 13** PSQ and PQ retrieval times for the sample retrieval example given in Fig. 9

It is also observed that the real PSQ retrieval times are close neighbourhood (i.e.  $\delta t_w < 0.5$  s) of  $t_p = 5$  s (user-defined period) value. One typical example showing PSQ arrival times for the PQ example shown in Fig. 9 is plotted in Fig. 13.

#### 4 Conclusions

In this paper we have proposed a simple, yet efficient query technique, called Progressive Query (PQ). PQ is particularly suited to large-scale multimedia databases. We believe that the user interaction with the active query process has the utmost importance and provides significant advantages in terms of user (and CPU) time and I/O accesses. Therefore, PQ is primarily developed to provide periodic and faster retrievals along with the ongoing query process. From this the user can get an idea about the current status of the query, immediately evaluate the available retrieval results and if satisfactory results are already achieved, the user can then halt the query process without wasting further time. We have confirmed this with a significant number of experiments.

As an efficient retrieval technique, PQ can be especially useful for databases lacking an efficient indexing structure. Naturally the exhaustive-search-based query operation, NQ, on such databases requires significant and sometimes even unfeasible retrieval times. Approvingly, an important objective achieved with the proposed PQ technique is that it avoids such implementation drawbacks, which NQ encounters. Experimental results show that PQ is not affected from such drawbacks and currently has no limitations with respect to system configuration.

PQ can also be applied to indexed databases. Once a query path is formed within the indexing structure, the most relevant items can be retrieved first. Moreover, we can foresee that PQ can provide a dynamic basis for relevance feedback in such a way that the user can employ relevance feedback during the query process and the incoming progressive retrievals can achieve a better retrieval performance. Since this is a natural consequence of user interaction capability, it was the primary inspiration of developing an interactive query technique such as PQ. In our future work, we plan to develop an efficient, self-learning and MAM-based indexing structure, which is designed to

provide optimal performance for PQ along with an effective relevance feedback mechanism under MUVIS framework.

#### 5 References

- 1 Kiranyaz, S., Caglar, K., Guldogan, O., and Karaoglu, E.: 'MUVIS: A multimedia browsing, indexing and retrieval framework'. Proc. Third Int. Workshop on Content Based Multimedia Indexing, CBMI, Rennes, France, 2003
- 2 MUVIS. <http://muvis.cs.tut.fi>
- 3 Pentland, A., Picard, R.W., and Sclaroff, S.: 'Photobook: tools for content based manipulation of image databases', *Proc SPIE (Storage and Retrieval for Image and Video Databases II)*, 1994, **2185**, pp. 34–37
- 4 Smith, J.R., and Chang: 'VisualSEEK: a fully automated content-based image query system'. ACM Multimedia, Boston, Nov. 1996
- 5 Virage. [URL:www.virage.com](http://www.virage.com)
- 6 Chang, S.F., Chen, W., Meng, J., Sundaram, H., and Zhong, D.: 'VideoQ: an automated content based video search system using visual cues'. Proc. ACM Mult., Seattle, 1997
- 7 Bentley, J.L.: 'Multidimensional binary search trees used for associative searching', *Commun. ACM*, 1975, **18**, (9), pp. 509–517
- 8 Guttman, A.: 'R-trees: a dynamic index structure for spatial searching'. Proc. ACM SIGMOD, 1984, pp. 47–57
- 9 Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B.: 'The R\*-tree: An efficient and robust access method for points and rectangles'. Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322–331
- 10 Sellis, T.K., Roussopoulos, N., and Faloutsos, C.: 'The R+-tree: A dynamic index for multi-dimensional objects'. Proc. 13th Int. Conf. on Very Large Data Bases, September 1987, pp. 507–518
- 11 Lin, K., Jagadish, H.V., and Faloutsos, C.: 'The TV-tree: an index for high dimensional data', *VLDB J.*, 1994, **3**, (4), pp. 517–543
- 12 Berchtold, S., Keim, D.A., and Kriegel, H.-P.: 'The X-tree: An index structure for high-dimensional data'. Proc. 22th VLDB Conf., 1996
- 13 White, D., and Jain, R.: 'Similarity indexing with the SS-tree'. Proc. 12th IEEE Int. Conf. On Data Engineering, 1996, pp. 516–523
- 14 Katayama, N., and Satoh, S.: 'The SR-tree: an index structure for high-dimensional nearest neighbor queries'. Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Arizona, US, 1997, pp. 69–380
- 15 Wang, H., and Perng, C.-S.: 'The S<sup>2</sup>-tree: an index structure for subsequence matching of spatial objects'. 5th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD), Hong Kong, 2001
- 16 Chakrabarti, K., and Mehrotra, S.: 'The hybrid tree: An index structure for high dimensional feature spaces'. Proc. Int. Conf. on Data Engineering, February 1999, pp. 440–447
- 17 Sakurai, Y., Yoshikawa, M., Uemura, S., and Kojima, H.: 'The A-tree: an index structure for High-Dimensional spaces using relative approximation'. Proc. 26th Int. Conf. on Very Large Data Bases, September 2000, pp. 516–526
- 18 Berchtold, S., Bohm, C., Jagadish, H.V., Kriegel, H.-P., and Sander, J.: 'Independent quantization: an index compression technique for high-dimensional data spaces'. Proc. 16th Int. Conf. on Data Engineering, San Diego, USA, 2000, pp. 577–588
- 19 Berchtold, S., Böhm, C., and Kriegel, H.-P.: 'The pyramid-technique: towards breaking the curse of dimensionality'. Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Washington, US, 1998, pp. 142–153
- 20 Fonseca, M.J., and Jorge, J.A.: 'Indexing high-dimensional data for content-based retrieval in large databases'. Eighth Int. Conf. on Database Systems for Advanced Applications (DASFAA), Kyoto, Japan, 2003, pp. 267–274
- 21 Yianilos, P.N.: 'Data structures and algorithms for nearest neighbor search in general metric spaces'. Proc. Fourth Ann. ACM-SIAM Symp. on Discrete Algorithms, Austin, Texas, US, 1993, pp. 311–321
- 22 Bozkaya, T., and Ozsoyoglu, Z.M.: 'Distance-based indexing for high-dimensional metric spaces'. Proc. ACM-SIGMOD, 1997, pp. 357–368
- 23 Brin, S.: 'Near neighbor search in metric spaces', *VLDB J.*, 1995, pp. 574–584
- 24 Ciaccia, P., Patella, M., and Zezula, P.: 'M+-tree: an efficient access method for similarity search in metric spaces'. Int. Conf. on Very Large Databases (VLDB), Athens, Greece, 1997, pp. 426–435
- 25 Traina, Jr, C., Traina, A.J.M., Seeger, B., and Faloutsos, C.: 'Slim-trees: high performance metric trees minimizing overlap between nodes'. EDBT, Konstanz, Germany, 2000, pp. 51–65
- 26 Zhou, X., Wang, G., Yu, J.X., and Yu, G.: 'M+-tree: a new dynamical multidimensional index for metric spaces'. Proc. Fourteenth Australasian Database Conf. on Database Technologies, Adelaide, Australia, 2003, pp. 161–168
- 27 Cheikh, F.A., Cramariuc, B., and Gabbouj, M.: 'Relevance feedback for shape query refinement'. Proc. IEEE Int. Conf. on Image Processing ICIP, Barcelona, Spain, 2003
- 28 Rui, Y., Huang, T.S., and Mehrotra, S.: 'Relevance feedback techniques in interactive content-based image retrieval'. Proc. IS&T and SPIE Storage and Retrieval of Image and Video Databases VI, San Juan, Puerto Rico, June 1997, pp. 762–768
- 29 The Open Video Project. <http://www.open-video.org/>
- 30 Partio, M., Cramariuc, B., Gabbouj, M., and Visa, A.: 'Rock texture retrieval using gray level co-occurrence matrix'. Proc. 5th Nordic Signal Processing Symp., Oct. 2002