Performance Analysis of Audio-Based Similarity Queries Using Clustering [*]

Latifur Khan, Mohammad Alshayeji, Ning Jiang, Cyrus Shahabi and Dennis McLeod
Integrated Media Systems Center and
Computer Science Department
University of Southern California
Los Angeles, California 90089
[latifurk, malshaye, ningjian, cshahabi, mcleod]@cs.usc.edu

Abstract

Many multimedia applications require the storage and retrieval of non-traditional data types such as audio, video and images. One important functionality required by these applications is the capability to find objects in a database that are *similar* to a given object. The comparison algorithms for multimedia data types are typically computationally expensive. Therefore, the performance of similarity queries can be improved significantly by reducing the number of invocations of these comparison algorithms. In this paper, we propose the utilization of clustering techniques in order to reduce the number of invocations of comparison algorithms.

Although clustering improves the performance of similarity queries, it might introduce inaccuracy in the results. We propose a family of similarity query execution techniques to strike a compromise between accuracy and performance. To evaluate the performance (i.e., query response time) and accuracy (i.e., precision and recall) of our similarity query execution techniques, we built an experimental setup using Informix OR-DBMS and AIR audio content-extraction algorithm. Our results demonstrate a significant improvement in performance while the accuracy of the results is maintained at a reasonable level.

1 Introduction

For decades, access methods such as index structures have been used to improve the performance of traditional database systems. B+-trees, for example, can significantly improve the performance of queries containing equality and/or linear range predicates. However, non-traditional (multimedia) data types such as image, audio or video, introduce new challenges that can not be met by conventional indexing techniques. These challenges can be addressed, however by designing new access methods specific to each new data type (e.g., R-trees for spatial data types). The problem with this approach is that whenever a new data type is introduced, the DBMS vendor not only needs to develop a new access

[*]This research was supported in part by gifts from Informix, Intel, NASA/JPL Contract no. 961518 and NSF grants EEC-9529152 (IMSC ERC) and MRI-9724567.

method (e.g., a tree structure) but also must integrate it with concurrency control, crash recovery and other database primitives. To remedy that, a Generalized Search Tree (GiST) has been proposed in [17]. GiST provides a unified data structure that can be extended to meet the requirement of new data types. In some cases, access methods are needed to improve the performance of systems employing black-box functions. However, it is not possible to build an effective index structure without knowing the details of these functions. In this paper, we investigate clustering as an alternative access method for such systems.

One of the most common methods to support multimedia data within conventional database systems is to extend the system with third party components in order to facilitate the access and evaluation of multimedia data types (more on this in Sec. 3). The Musclefish Audio Information Retrieval (AIR) DataBlade module [1, 6], for example, can be used with the Informix Universal Server in order to integrate sound as a new data type into the database system. AIR can support the search and retrieval of digitized sound in very large databases based on the characteristics of the audio rather than keywords. In general, this type of environment can be characterized as follows:

a: The third party component provides for a similarity function (i.e., a function that can take two objects and estimate how similar they are to each other).

b: The similarity function is computationally expensive.

c: The details of the similarity function are not provided.

Without knowing the details of the similarity function it is virtually impossible to implement an effective indexing structure that can improve the performance of similarity queries. Moreover, the AIR DataBlade provides a representation function (termed *makemodel*) that given a set of audio objects can generate a model that represents these objects. We propose to utilize these two black-box functions to cluster database objects [22]. Access methods can then utilize these clusters to reduce the need to invoke the computationally expensive similarity function thus improving the performance of similarity queries. Although we focus on audio similarity queries, all the techniques we present here are generalizable to other media types (e.g., image, video) with minor modifications.

To illustrate our approach, suppose a database contains n multimedia objects. To support a *selection* query (i.e., finding all the objects similar to a target object o), the system requires to invoke the similarity function n times. Now suppose we can partition the database (off-line) into m clusters of k objects[1] ($n = m \times k$) where the objects in a cluster are more similar to each other than to those belonging to other clusters. Note that we can use the same provided similarity function to perform this clustering. Now, to support the selection query, we can compare o with only the representative

[1]To simplify, here we assume a uniform distribution of objects into clusters.

of each cluster. Therefore, the expensive similarity function is invoked m times as opposed to n times (a factor of k improvement). Trivially, we are sacrificing accuracy in order to improve performance. In this paper, hence, we describe a family of similarity query execution techniques to compromise between accuracy and performance depending on the requirements of an application domain. It is important to point out that none of these algorithms can fully guarantee accurate results. Therefore, these techniques are targeted toward applications that can tolerate some inaccuracy in their results, a fair assumption in multimedia applications.

The remainder of this paper is organized as follows. Sec. 2 provides an overview of some related works in the areas of clustering and multimedia query processing. In Sec. 3, we describe our multimedia database system architecture. In addition, we describe a couple of motivating applications (utilizing our system) that require efficient support of multimedia selection and join queries. Sec. 4 provides an overview of the clustering techniques evaluated in this paper. The family of our similarity query execution algorithms are presented in Sec. 5. In Sec. 6, two alternative implementations of multimedia join operation based on our similarity query execution algorithms are discussed. The performance of these techniques is evaluated in Sec. 7. Finally, Sec. 8 concludes the paper and provides an overview of our future plans.

2 Related Work

Clustering has been used in a number of areas such as statistics [18, 11], pattern recognition [14, 10] and machine learning [12], to name a few. In addition, there has been a significant amount of work in the area of content-based extraction [15, 1]. In this paper, we attempt to merge these two areas to develop techniques to improve the performance of similarity queries in multimedia databases without sacrificing the accuracy of the results. The goal of this work is not to investigate every clustering algorithm nor it is to propose new content-based extraction techniques. Rather, the goal is to show how to employ clustering algorithms that can utilize functionalities available in the content-based extraction arena to reduce the cost of the expensive similarity queries.

A number of studies [5, 16, 19] extend some indexing techniques to improve the performance of similarity queries for multimedia data with high-dimensional features. In [20], for example, algorithms for processing join over two collections of documents with attributes of textual types are presented. Similar to our work, each object (document) is represented by a vector which is compared in the text (similarity) join. However, in our environment, with a black-box similarity function, it is not possible to assign proper weights to each of the features in the feature vector representing a multimedia object. Without these weights, generating an effective high dimensional index (e.g., SS tree [26]) is not possible making clustering the natural substitute.

Recently, in [25], different hierarchical clustering techniques are described to facilitate similarity query efficiently in an image database system. They assume interpretation of feature vector is available to the system in order to generate the clusters. This assumption is not valid within our ORDBMS environment. In addition, they do not address query processing issues at the database level to support similarity and join queries.

3 System Architecture and Motivating Applications

There are three main approaches to develop multimedia database systems with content-based query capabilities. The first trend is to utilize an object-relational database system and extend it to support query and access to non-traditional data types such as images, audio and video. This can be achieved for example by using Datablades of Informix Universal Server, UDFs of NCR Teradata Object Relational database system, Cartridges of Oracle 8, or Extenders of IBM DB2. The second trend is to build special purpose systems from scratch to support content-based queries for only a single media type, such as QBIC [13] and Virage[3] for images, [21, 27, 9, 8, 23] for video, and [1, 6] for audio to name a few. Finally, the third approach is to develop special purpose systems for a specific application which might support content-based queries on multiple media types transparently from the user (e.g., Informedia [7] project at CMU). At the USC Integrated Media System Center (IMSC), we are building a system following the first approach.

We started with an object-relational database system, namely Informix Universal Server (IUS) v9.12 and then extended it with content-extraction algorithms developed by IMSC investigators (e.g., image content extraction, face recognition, semi-structured wrappers) as well as others (e.g., PPM image library, Musclefish audio content extraction, and CMU JANUS voice recognition toolkit). Finally, we designed and built the conceptual schema and the user interface required by a specific application (namely, Media Immersion Environment or MIE for short) to integrate all these and hide the details from users. Our system can be reshaped, with little effort, to be utilized for supporting query and access to other multimedia applications.

MIE database system is currently being used to store and retrieve tele-conferencing sessions. Its current architecture provides access to three organizational repositories: a database server (IUS v9.12), an MIE Real-Time File Server (to support storage and retrieval of continuous media), and an Information Mediator [4]. This distributed repository can support the storage, query and retrieval of different media types such as audio, video, images and text. A middleware is used to glue these repositories in order to provide a unified access from the client to these sources. This middleware is implemented in

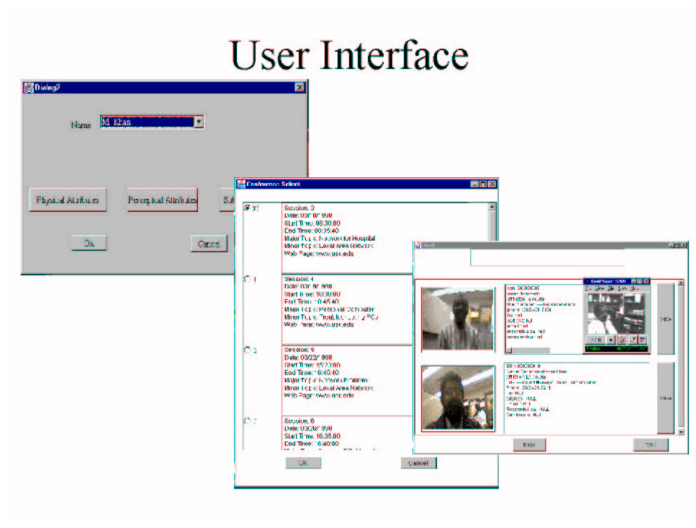

Figure 1: User Interface

Java language and it is loaded to the client space at run-time. The main advantage of this design is that the system operates in 2-tier mode at every instance of time and hence makes the performance superior to 3-tier architectures employed by industry.

To search for a stored tele-conferencing session, a user can submit queries to MIE database system such as: *find all the sessions that the IMSC director, "Nikias", participated in.* The user interface (Fig. 1) can be used to submit such a query. To support this query, the system utilizes three relations of the MIE database schema: *Samples*, *Clips* and *Sessions* relations. The *Samples* relation contains the names of all the IMSC staff, students and faculty and a sample audio clip for each participant. The *Clips* relation stores the audio clips for each participant in the actual tele-conferencing sessions where each audio clip is associated with a session_id to which the clip belongs. Finally, the *Sessions* relation contains information about each session such as time, date, duration and topic of a session. Fig. 2 shows the SQL equivalent of the above query (the *AudioCompare* function will be discussed later in Sec. 3.1). The system displays the result of the query in the manner shown in Fig. 1. The user can then select and play the session that she is interested in.

We plan to use the MIE database system in other applications such as those in the entertainment industry. The database may store a library of sound effects in a relation *Sound_Effects* that contains a textual description of each sound effect such as "door slam" or "gun shot" as well as audio samples of these sound effects. Subsequently, a relation *Terminator* may store all the sound effects used in the movie "Terminator". MIE database can then be used to answer a query of the form: *what is the number of gun shots in "Terminator"*. Since the sound effects are not unique (i.e., there might be more than one sample gun shot in the relation Sound_Effects), the above query can be executed by *joining* the two relations on the clip-mode attribute as shown in Fig. 3. In order for the system to execute such queries as in Figs. 2 and 3, it has to invoke the computationally expensive *AudioCompare* function

```
**select** s.*
**from** Sessions s
**where** s.session_id in
        (**select distinct** c.session_id
        **from** Clips c
        **where** AudioCompare((**select** m.clip_Model
                **from** Samples m
                **where** m.Participant='Nikias'),
                c.clip_analysis)≤ QT)
```

Figure 2: Tele-conferencing Example

many times.

```
**select count** (m.movie_id)
**from** Sound_Effects s, Terminator m
**where** s.description = 'gun shot'
**and** AudioCompare(m.clip_mode, s.clip_mode)≤ QT
```

Figure 3: Entertainment Industry Example

3.1 AIR

In order for the MIE database system to support the audio-based queries discussed in Sec. 5, it utilizes the Musclefish Audio Information Retrieval (AIR) DataBlade module [1, 6]. AIR can support the search and retrieval of digitized sound based on its content. To achieve this, audio clips are statistically analyzed by AIR via the function *AudioAnalyze* and an *Analysis record* is generated for each clip. The Analysis record is basically a set of 141 important *features* extracted from the clip. A useful function of AIR for our clustering purpose is its *AudioMakeModel* that can take a set of one or more Analysis records as input and generate a representative for their corresponding clips. The representative, termed *Model*, can then be used along with the clip analysis in the *AudioCompare* function to perform content-based comparisons. *AudioCompare* returns a floating point number indicating the level of similarity between audio clips (smaller values determine higher levels of similarity).

4 Overview of Some Clustering Techniques

The first step in our approach is to partition n (audio) objects into m clusters where $m < n$. There are many clustering techniques proposed in the literature for achieving this objective. We have decided to evaluate the effectiveness of a number of these clustering algorithms for our approach. Due to lack of space and since the focus of this paper is not on a comprehensive evaluation of clustering algorithms, we report on one or more representative techniques that we chose to incorporate into our approach for each of the following three main classes.

4.1 General Clustering Algorithms

In general, all clustering techniques employ a similarity measure to evaluate the distance among objects to be clustered. The objects which are closer in distance are then grouped into a cluster. Since each object has a number of *attributes* or *features*, the distance computation should take these attributes into account when evaluating the similarity between objects. Most of the clustering techniques, in their basic form, assign equal weights to each attribute when calculating/estimating the similarity between two objects. As one of the representatives of this class of algorithms, we selected the well known K-means clustering algorithm. K-means uses the Euclidean distance as a similarity measure. The K-means strives to generate a nearly optimal partition taken the number of clusters as input. First, an initial partition for the given cluster number is built. Later, the partition is improved iteratively keeping the same number of clusters.

Our second representative is the Clique [2] algorithm. Clique is known to be an effective *high-dimensional* clustering technique, appropriate for most of the multimedia data types, including audio, where each object is identified by a large number of attributes/features (e.g., 147 features with AIR). However, out of the many attributes associated with each audio clip, only a small set is used in the similarity measurement. Therefore, as we will show, a more effective clustering can be obtained by considering only the attributes that are actually used in the similarity measurement.

4.2 Similarity-Based Clustering Algorithms

One problem with general clustering techniques is that they treat attributes equally, that is, they assign equal weights (or importance) to all the features. However, some of these techniques can be tailored to the needs of a specific application by assigning different weights to different features. We classify these adjustable clustering algorithms into the class of similarity-based clustering techniques. Although this class seems like an attractive solution to be incorporated into our approach, it is inconsistent with the characteristics of our assumed environment (see Sec. 1). Recall that we are looking for a general purpose technique to improve the performance of similarity queries at the database system level for any third-party provided similarity function. Hence, the information about the weights of the attributes considered in the third-party similarity function (e.g., Musclefish AIR DataBlade) is not available to us, making it impossible to assign proper weights to these attributes for clustering purposes. To overcome this limitation, however, we altered the K-means algorithm to use the AudioCompare function provided by Musclefish, instead of its basic Euclidean distance function, as the similarity measure. It is important to note that this alteration is only applicable to those clustering techniques that employ some stand-alone similarity (distance) functions in their algorithms.

4.3 Threshold-Based Clustering Algorithms

We also decided to investigate a threshold based clustering algorithm. Threshold-based clustering limits the size of a cluster (i.e., the maximum similarity/distance among the members of a cluster) to a known value. Limiting the distance among cluster members allows us to develop efficient algorithms to execute similarity queries as we show in Sec. 5.

We altered a threshold-based clustering algorithm proposed in [24] to incorporate AudioCompare as the distance function used by the algorithm (similar to the modification we made to K-means in Sec. 4.2). Given a set of N sample clips $x_1, x_2, \cdots, x_N$ and a nonnegative cluster threshold CT, the algorithm, termed Simple Threshold Clustering (STC), starts by randomly selecting the first cluster representative z_1 from the N sample clips. For convenience, we may choose $z_1 = x_1$. Next, STC computes the AudioCompare distance D_{21} between x_2 and z_1. If this distance exceeds CT, a new cluster with $z_2 = x_2$, is created. Otherwise, x_2 is assigned to the domain of cluster z_1. If the latter is true, when considering x_3, the AudioCompare distances D_{31} and D_{32} (between x_3 and z_1 as well as x_3 and z_2) are computed. If both D_{31} and D_{32} are greater than CT, a new cluster with $z_3 = x_3$, is created. Otherwise, x_3 is assigned to the domain of the cluster to which it is closest. This procedure repeats until all N clips are exhausted.

5 Support of Selection Queries under Clusters

Once we employ one of the techniques described in Sec. 4 to cluster a set of multimedia objects (or a relation), there are alternative ways to execute a selection query on those clusters. We define *selection* query to have the general form of: "Find objects similar to a reference object within a threshold QT". This query is analogous to a selection operation performed on relational databases with a graded level match.

In this section, we describe three techniques to execute a selection query on clusters. With these techniques, we attempt to strike a compromise between the performance of the query (i.e., the number of operations required to execute the query), and the accuracy/quality of the results (i.e., how close is the result to a regular selection query executed in the absence of clusters). The first technique is the most efficient and the last is the most accurate.

To evaluate the accuracy of each technique, we employ standard Information Retrieval accuracy measures such as Recall and Precision which are defined as follows:

$$Recall = \frac{Ret \bigcap Rel}{Rel} \tag{1}$$

$$Precision = \frac{Ret \bigcap Rel}{Ret} \tag{2}$$

Where *Rel* is the set of relative objects (i.e., objects that would be retrieved by a regular selection query) and *Ret* is the set of retrieved object (i.e., objects that are actually retrieved by the clustered selection query).

To describe the alternative query execution techniques, the audio database is described with the following self-explanatory schema: *Sound_Clips(clip_id, clip_model, clip_analysis, file location).* To assign clips to clusters, a new attribute *cluster_id* is added to the *Sound_Clips* relation. Moreover, a new relation *Clusters(cluster_id, cluster_model)* is created to facilitate the comparison of the query reference to the clusters' representatives.

5.1 *Compare* Algorithm

```
select S.clip_id
from Sound_Clips S, Clusters C
where S.cluster_id = C.cluster_id
and AudioCompare
    (C.cluster_model, Reference_Analysis) ≤ QT
```

Figure 4: *Compare*

Depending on the type of application, the user may be more concerned with the turn-around time and willing to sacrifice accuracy in favor of performance. The *Compare* algorithm(Fig. 4), in this case, can execute a selection query by comparing the analysis of the reference provided by the query to the model of each cluster in the database. When the similarity between the two is less than or equal to the threshold, *Compare* outputs all the cluster elements. By comparing the reference only to the clusters' models, as opposed to comparing it to every element in the database, *Compare* significantly reduces the number of invocations of the expensive AudioCompare operation.

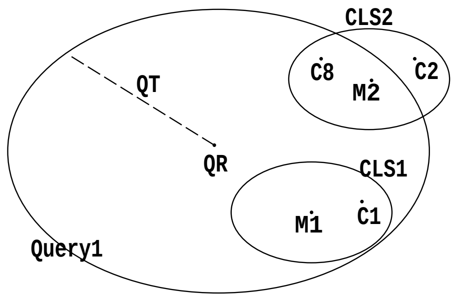

Figure 5: Selection Query with Clustering (Example 1)

Compare, however, suffers from serious limitations when it comes to the accuracy of the query results. Briefly, *Compare* might result in *false hits*, that is it might wrongly identify an object as a match with the reference object. To illustrate consider Fig. 5 which depicts a clustered database. In Fig. 5, clip $C1$ is a member of cluster $CLS1$ whose model is $M1$. Similarly, clips $C2$ and $C8$ are members of cluster $CLS2$ whose model is $M2$. Here, QR is the reference object and QT is the query threshold for a selection query Q. Since the models of both clusters $CLS1$ and $CLS2$ fall within the query threshold QT boundary, *Compare* identifies all the elements of both $CLS1$ and $CLS2$ as matches with QR. Note that even though clip $C2$ does not fall within the boundary of the query threshold, its cluster model does. Therefore, it is incorrectly identified by *Compare* as a match resulting in a false hit.

```
select distinct S1.clip_id
from Sound_Clips S1
where exists ( select * from Clusters C
               where S1.cluster_id = C.cluster_id
               and AudioCompare
               (C.cluster_model, Reference_Analysis) ≤QT
               and exists
               (select * from Sound_Clips S2
               where S1.clip_id = S2.clip_id
               and S2.cluster_id=C.cluster_id
               and AudioCompare
               (S2.clip_model, Reference_Analysis) ≤QT))
```

Figure 6: *Compare&Expand*

5.2 *Compare&Expand* Algorithm

To improve the accuracy of *Compare* by eliminating false hits, we propose the *Compare&Expand* algorithm (Fig. 6). Instead of blindly identifying all the elements of the clusters that fall inside the boundary of the query threshold as matches, *Compare&Expand* expands each of the matched clusters and compares all of their elements to the reference object. As a result, when expanding cluster $CLS2$ in Fig. 5, *Compare&Expand* recognizes that clip $C2$ is outside the query threshold boundary thus dropping it from the query result. By Comparing the SQL queries in Fig. 4 and 6, it is obvious that the performance of *Compare&Expand* is worse than that of *Compare* due to the extra invocations of AudioCompare in Fig. 6. However, *Compare&Expand* eliminates false hits thus improving the accuracy (precision) of the result.

5.3 *Compare*&Expand* Algorithm

Even though it eliminates false hits, *Compare&Expand* may result in false drops, that is, it may fail to identify all the objects that match the reference object. To illustrate, in Fig. 7, *M3* which is the model of cluster $CLS3$ falls outside the boundary of QT. As a result, the members of $CLS3$, are not considered as matches, thus resulting in a false drop (i.e., $C3$). To reduce the number of false drops, we propose *Compare*&Expand*. To reduce the probability of missing an object due its cluster's model falling outside the boundary of QT, we extend the domain of clusters considered by the similarity query by a constant L. Subsequently, we expand all those clusters whose model fall

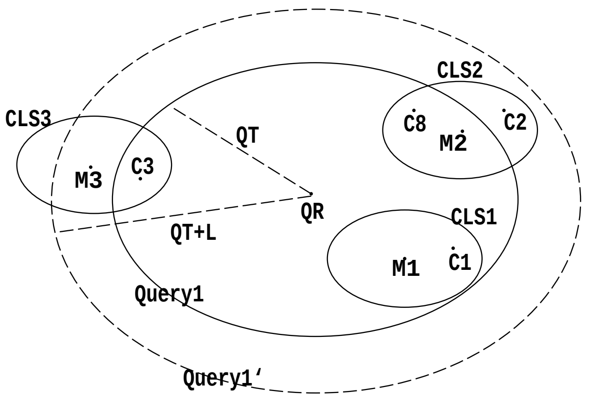

Figure 7: Selection Query with Clustering (Example 2)

within the boundary of $QT' = QT + L$, instead of QT. The objects of these clusters, however, are not identified as matches unless they fall within the boundary of QT. In this case, a proper value of L results into the consideration of $CLS3$, in Fig. 7, hence including $C3$ in the result. In this example, the value of L was large enough to avoid an otherwise false drop. In general, however, selecting the value of L is not a trivial task. Selecting a large value of L results in considering a larger number of clusters and thus defeating the purpose of clustering. A small value of L, on the other hand, may not be effective enough in reducing the number of false drops.

```
select distinct S1.clip_id
from Sound_Clips S1
where exists
        (select * from Clusters C
        where S1.cluster_id = C.cluster_id
        and AudioCompare
        (C.cluster_model, Reference_Analysis) ≤QT+L
        and exists
            (select * from Sound_Clips S2
            where S1.clip_id = S2.clip_id
            and S2.cluster_id=C.cluster_id
            and AudioCompare
            (S2.clip_model, Reference_Analysis ≤QT))
```

Figure 8: $Compare^{*}\&Expand$

Using a threshold based clustering technique (see Sec. 4.2), in this case, can be advantageous. Knowing that the distance between the representative of a cluster and all of its members is less than or equal to the cluster threshold CT, provides us with a good default value for $L = CT$. Although using this value for L does not guarantee the elimination of false drops, it can significantly reduce the probability of their occurrences. An L value that can guarantee the elimination of false drops can only be computed through the full knowledge of the details of similarity function AudioCompare, which is inconsistent with our assumed environment. Note that using other non-threshold based clustering algorithms can further complicate this problem since these algorithms do not have a bound on the distance between the model of a cluster and its members. We are currently investigating more sophisticated methods to compute the value of L without knowing the details of the similarity function AudioCompare.

6 Support of Join Queries under Clusters

Two out of the three selection query execution algorithms we described in Sec. 5 require an extra *expansion* step in order to eliminate false hits. In case of *join* queries, it becomes important to execute the expansion step in an efficient manner. The idea is that an expanded cluster can be kept in memory while all the tuples in the first relation are compared with the objects belonging to the expanded cluster. In this section, we present two algorithms for the implementation of clustered audio join. Here, page based algorithm is considered. Since I/O cost is the dominating cost, we focus on minimizing the number of random I/O access as our optimization criteria.

6.1 Nested Loop (NL)

In this section, we describe the basic clustered join which is similar to the classical nested loop join algorithm. To assist in the discussion of the algorithm, we start by stating our assumptions. Suppose that we are interested in joining the two relations, R_1 and R_2. Let the audio clips of R_2 be clustered into m clusters, and hence we denote the clustered relation R_c as a set of clusters, i.e., $R_c = \{x_1, x_2, \cdots, x_m\}$. Since we are focusing on page based algorithms, let K be the size (in pages) of the available memory buffer (output buffer page(s) are excluded from K), and N_1, N_2, N_c, are the sizes in number of pages of the relations R_1, R_2 and R_c, respectively. Moreover, assume that R_1, R_2, and R_c use K_1, K_2 and K_c buffer pages respectively where $K_1 \leq N_1$, $K_2 \leq N_2$, and $K_c \leq N_c$. Without loss of generality, we also assume $K \geq 3$.

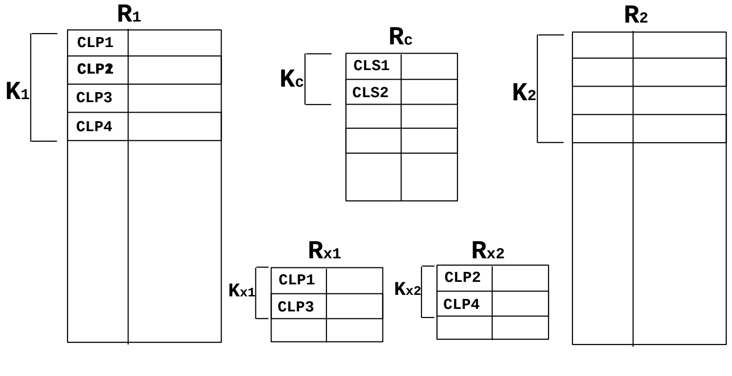

Figure 9: Join operation with clusters

Fig. 10 depicts the NL algorithm. In the algorithm, R_c is used in the outermost loop[2]. As a result,

[2]By the end of this section we also discuss the case where R_1 is the outer relation.

R_c is read only once. To minimize I/O cost, the algorithm compares the memory resident tuples of R_c with all the tuples of R_1. Therefore, for each K_c pages of R_c residing in memory, the entire R_1 needs to be read. However, since it is a page based nested loop, for each K_1 pages of R_1 containing at least one tuple matching a cluster in R_c, the entire R_2 needs to be read (the expansion step). The I/O cost of this nested loop algorithm can hence be computed as:

$$I/O_{NL} = N_c + \lceil \frac{N_c}{K_c} \rceil \times N_1 + \sum_{x_i \in R_c} h_{x_i} \times \lceil \frac{N_c}{K_c} \rceil \times N_1 \times N_2 \qquad (3)$$

Where h_{x_i} is the hit ratio of N_1 for x_i which is the number of pages out of N_1 which contain at least one tuple similar to the cluster x_i.

```
for each $K_c$ pages $Q$ of $R_c$
    for each $K_1$ pages $P$ of $R_1$
        for each tuple $x_i$ in $Q$
            let flag = false
            for each tuple $y$ in $P$
                compute similarity between $y$ & $x_i$
                if similarity within the threshold
                  then flag = true break
            if (flag=true)
              then
                for each $K_2$ pages $S$ of $R_2$
                    compute similarity
                    between tuples from $P$ & $S$
                    if similarity within the threshold
                      then output tuples
```

Figure 10: Nested Loop(NL)

It can be shown that the lower bound of I/O cost in Eq. 3 is $K1 = min(N_c, K - 2)$. In other words, the outer relation should use as many buffer pages as needed (no more than N_c buffer pages). Similarly, when R_1 is the outer relation, the I/O cost is:

$$I/O_{NL} = N_1 + \lceil \frac{N_1}{K_1} \rceil \times N_c + \sum_{x_i \in R_c} h_{x_i} \times \lceil \frac{N_1}{K_1} \rceil \times N_c \times N_2 \qquad (4)$$

The lower bound of I/O cost in Eq. 4 is $K1 = min(N_1, K - 2)$.

6.2 Staged Nested Loop (SNL)

In this section, we present an alternative clustered join algorithm termed *Staged Nested Loop (SNL)*. This algorithm attempts to minimize the I/O cost by introducing a preparation stage (Fig. 11). The NL algorithm suffers from what we term as the multiple expansions problem. In Fig 9, for example, if

```
for each $K_c$ pages Q of $R_c$
    for each $K_1$ pages P of $R_1$
        for each tuple $x_i$ in $Q$
            for each tuple y in $P$
                compute similarity between $x_i$ & $y$
                if similarity within the threshold
                    then insert tuple $y$ into $R_{x_i}$
    for each $K_{x_i}$ pages $X$ of $R_{x_i}$
        for each $K_2$ pages S of $R_2$
            for each tuple y in $X$
                for each tuple z in $S$
                    compute similarity between $y$ & $z$
                    if similarity is within the threshold
                        then output tuple (y,z)
    delete all tuples from $R_{x_i}$
```

Figure 11: Staged Nested Loop (SNL)

a clip in K_1 (say CLP1) is identified to be similar to a cluster in K_c (say x_1), the algorithm expands x_1 resulting to a full pass on R_2. Another clip in K_1 matching x_1 may use the result of this expansion, that may be resident in K_2, and hence avoiding another pass on R_2. Since page based algorithms are geared toward minimizing I/O, the algorithm will exhaust the content of K_1 before replacing it. Therefore, cluster x_1 will be compared to all the clips in K_1 before considering x_2. Subsequently, when considering x_2, it is possible that another clip (say CLP2) is found to be similar to x_2 resulting in the expansion of x_2 and another pass on R_2. This expansion may result in the replacement of the pages containing the member clips of cluster x_1 in K_2 with that of x_2. After exhausting all the clips in K_1, K_1 is replaced. Once again a x_1 is compared to the new clips in K_1. A clip-cluster match may result in the replacement of the pages containing the member clips of cluster x_2 with that of x_1. This multiple expansions is possible for every cluster in K_c and for every replacement of K_1. Clearly this will result in a high I/O cost ($m * \lceil (N_1/K_1) \rceil$ cluster expansions in the worst case).

To remedy this, we propose the SNL algorithm. With SNL, an aggregation stage is employed to gather all the clips of R_1 that are similar to a cluster x_i. In this stage, a temporary relation R_{x_i} is created for each cluster x_i in K_c (Fig. 9). When the algorithm determines that a clip in K_1 is similar to a cluster x_i in K_c the clip is inserted into R_{x_i}. After exhausting the clusters in K_c, R_{xi} is joined with R_2. The content of K_c is then replaced and the temporary relations R_{x_i} are deleted to make space for new temporary relations. As a result of grouping matched clips and delaying the expansion, each cluster in R_c can at most observe one expansion (assuming that the members of a cluster can fit in memory, a realistic assumption in this case), hence eliminating the multiple expansions problem. Therefore, SNL needs only m expansions to perform the join. However, to achieve this, the algorithm incurs the extra cost of inserting these clips into the temporary relations. Note that the cost of a cluster expansion can be improved by indexing R_2 on cluster_id to avoid a full pass for every expansion. However, this improvement contributes equally to both algorithms and thus is not considered here.

In the general case, the I/O cost of SNL can be calculated as:

$$I/O_{SNL} = N_c + \lceil \frac{N_c}{K_c} \rceil \times N_1 + \sum_{x_i \in R_c} (I/O_{INSERT}(s_{x_i}, R_1) + \lceil \frac{N_c}{K_c} \rceil \times N_{x_i} \times N_2) \tag{5}$$

Where s_{x_i}, the selection factor between relation R_1 and x_i, equals to the ratio of tuples of R_1 that are similar to x_i over the total number of tuples in R_1. N_{x_i} is the size of the relation R_{x_i}.

6.3 Analysis

We show analytically for what cases SNL is better than NL. For this, let B_{x_i} be the number of blocks of R_1 where clips are similar to cluster, x_i and hence, η_{x_i} is defined by,

$$\eta_{x_i} = \frac{1}{B_{x_i}} \tag{6}$$

Where B_{x_i} is the number of blocks of R_1 where clips are similar to tuple, x_i. Here, η_{x_i} controls the distribution of similar tuples among the N_1/K_1 blocks of R_1. When similar tuples are concentrated in one block then $\eta_{x_i} = 1$. Moreover, if $\eta_{x_i} \ll 1$ (i.e., $h_{x_i} \gg 1$), then similar tuples are scattered among the blocks of N_1. It is important to note that h_{x_i} can be defined in terms of N_{x_i}. We assume selected tuples of R_1 which are similar to x_i constitute N_{x_i} pages. Therefore,

$$h_{x_i} = \frac{N_{x_i}}{\eta_{x_i} \times N_1} \tag{7}$$

The write operation is usually 2 to 3 times as expensive as the read operation. $I/O_{INSERT}(s_{x_i}, R_1)$ is characterized by writing N_{x_i} pages into disk. Therefore, the cost of insertion can be calculated as:

$$I/O_{INSERT}(s_{x_i}, R_1) = w \times N_{x_i} \tag{8}$$

Replacing for Eq. 7, and 8 into Eq. 3, and 5, Eq. 3 becomes

$$\begin{aligned} I/O_{NL} = N_c + \lceil \frac{N_c}{K_c} \rceil \times N_1 + \sum_{x_i \in R_c} \lceil \frac{N_c}{K_c} \rceil \times \frac{N_{x_i} \times N_1 \times N_2}{\eta_{x_i} \times N_1} \\ = N_c + \lceil \frac{N_c}{K_c} \rceil \times N_1 + \sum_{x_i \in R_c} \lceil \frac{N_c}{K_c} \rceil \times \frac{N_{x_i} \times N_2}{\eta_{x_i}} \end{aligned} \tag{9}$$

Moreover, Eq. 5 becomes,

$$I/O_{SNL} = N_c + \lceil \frac{N_c}{K_c} \rceil \times N_1 + \sum_{x_i \in R_c} (w \times N_{x_i} + \lceil \frac{N_c}{K_c} \rceil \times N_{x_i} \times N_2) \tag{10}$$

From Eqs. 9 and 10, we can conclude that an $\eta_{x_i} \ll 1$ favors the SNL algorithm. On the other hand, a $\eta_{x_i} = 1$ favors the NL algorithm. However, for SNL to be computationally effective, the value of η_{x_i} must be small enough to compensate for the cost of inserting similar tuples into temporary relations.

7 Performance Analysis

We start by describing our experimental setup and then we present our preliminary results. We have conducted a number of experiments to evaluate the effectiveness of our different algorithms as compared to traditional join. It is important to note that so far we have discussed selection queries for similarity. In contrast, results were reported for join queries. This is because we may consider a number of reference objects for selection queries as a separate relation. We also evaluated the performance of one of these algorithms (*Compare**&*Expand*) when applied to the three clustering techniques we discuss in this paper. First, we verified that our algorithms observe less response time as compared to traditional join for particular cluster and join thresholds. We reported the results only for SNL in order to avoid the multiple expansions problem (see Sec. 6).

Our experimental setup consists of an Informix Universal Server (IUS) which is an object-relational DBMS running on dual processor SUN Ultra Server 450. Each processor is 300 MHZ Ultra sparc II with 1 GB RAM. To support audio data type, AIR DataBlade module is incorporated into IUS. Each audio clip's *analysis* and *model* are inserted into the relation during the population of the relation. For the experimental purposes, we considered two relations, named *Sound_Clips* and *Sample_Clips* which consist of 206 and 4 audio clips, respectively. A relation *Clusters* is created from the relation *Sound_Clips* using our variation of the *STC* clustering technique (see Sec. 4.3)

In our experiments, we varied cluster threshold from 0.1 to 0.4. Clearly, smaller values of CT generate larger number of clusters, and hence increasing the response time of *Compare*. On the other hand, higher values of CT generate smaller number of clusters. Consequently, *Compare*&*Expand* and *Compare**&*Expand* may observe higher response time by expanding the matched clusters which may now have more clips. We only report the result for CT=0.3. This is because at CT=0.3, 19 clusters are generated, among which 206 clips are partitioned almost uniformly thus creating a very suitable testing environment. We assume the schema of *Sample_Clip* to be identical to that of *Sound_Clips* (see Sec. 5) and the primary key *clip_id* is indexed for both relations. The assumed join query is:

```
select S1.clip_id, S2.clip_id
from Sample_Clips S1, Sound_Clips S2
where AudioCompare
        (S2.clip_model, S1.clip_Analysis) ≤QT
```

7.1 Results

For the first set of experiments, we compared the query response time of different query execution algorithms with CT=0.3. In Fig. 12, the X-axis represents QT which is varied from 0.1 to 0.9 while the Y-axis represents the query response time in millisecond. *Compare*, *Compare&Expand* and *Compare*&Expand* and traditional join are represented by solid, dotted, dashed, and star lines respectively.

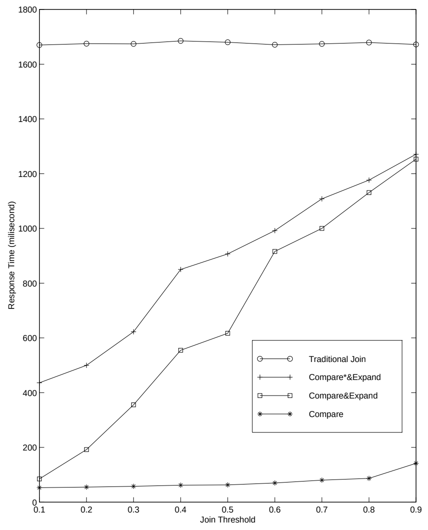

Figure 12: Response time of different algorithms for Audio-Based Join Queries

As discussed earlier, the three query execution algorithms compromise between accuracy and response time. Fig 12 shows the response time of the three algorithms as compared with traditional join. As expected, *Compare* observed the best response time since it only considers clusters' representatives (i.e., no expansion stage). *Compare* outperforms traditional join by a factor of 30. Moreover, both *Compare&Expand* and *Compare*&Expand* outperform the traditional join. However, unlike *Compare* and the traditional join and the response time of *Compare&Expand* and *Compare*&Expand* is a function of QT. In general, as the value of QT increases, the response time of *Compare&Expand* and *Compare*&Expand* increase due to higher number of matched clusters and hence an explosion of the expansion step. This explains why the performance of *Compare* is almost independent of the value of QT. Moreover, the number of invocations of AudioCompare is always equal to the product of the cardinalities of relations *Sample_Clips* and *Sound_Clips* with traditional join and hence, is independent of QT.

Compare&Expand and *Compare*&Expand* compensate for the higher response time by improving the accuracy of the results. As expected *Compare* never achieved a 100% precision due to the presence of false hits. This is because all the member clips of the selected clusters are projected into the result, even though some of these member clips may not be similar to the sample clip. These false hits, however, are discarded in the *expansion* phase of *Compare&Expand* and *Compare*&Expand* algorithms. Therefore, the latter two algorithms both achieved a 100% precision for all values of QT (Fig 13).

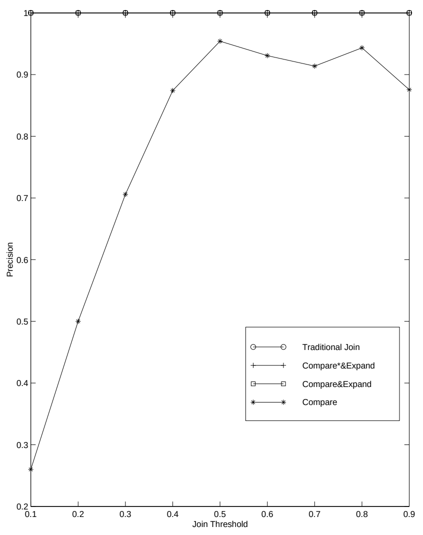

Figure 13: Precision of different algorithms for Audio-Based Join Queries

Both *Compare* and *Compare&Expand* suffer from false drops. *Compare*&Expand* attempt to eliminate this problem by increasing the number of clusters considered in the *expansion* phase. After experimenting with different values of L for *Compare*&Expand*, we observed that a L=CT guarantees 100% recall for our dataset. This is because at QT+L, all relevant clusters (and, clips) are retrieved (i.e., no false drops). While this cannot be generalized to all datasets, it shows that L=CT is a good heuristic. Clearly, L has an impact on query response time. Larger L selects more relevant clusters and consequently, the *expansion* cost will increase. The recall of the three algorithms as compared to traditional join is shown in Fig. 14.

Due to lack of space, we do not report the detail results of the comparison between different clustering techniques. Our results showed our variation of *STC* to be the most appropriate clustering technique. Even though, STC observed response time higher than both Clique and K-means, it showed a much better recall. STC had a perfect 100% recall, with about 40% and 60% recall for Clique and K-means respectively when employing *Compare*&Expand*. The member of the clusters generated by STC has a known maximum distance from the cluster representative which enabled *Compare*&Expand* to reach maximum recall with L=CT. The same, however, can not be said about the other two clustering techniques.

8 Conclusion and Future Research

We proposed clustering as an alternative access method in the absence of effective indexing structures for some non-traditional data types such as audio. We proposed the utilization of clustering techniques in order to reduce the number of invocations of comparison algorithms for similarity queries. Our approach is flexible enough to take advantage of any black-box comparison algorithm provided by a third-party vendor for object-relational database systems. We described three similarity query execution techniques to strike a compromise between accuracy and performance of clustered selection

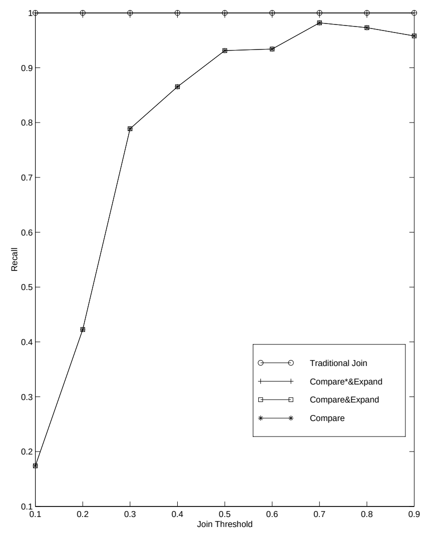

Figure 14: Recall of different algorithms for Audio-Based Join Queries

queries. Using our experimental setup, we demonstrated that while *Compare* results in the best performance in query response time, *Compare*&Expand* generates the most accurate results with 100% precision and recall. This accuracy which matches that of regular audio join, can be achieved with a significantly lower response time.

While the *Compare*&Expand* outperforms the traditional join by a factor of 3 (see Sec. 7 for the case where $CT = 0.3$, $JT = 0.1$, and $L = 0.2$). We also investigated clustered join queries for multimedia data types. Two alternative implementations were proposed and analytical models were provided to show when one is superior to the other.

We intend to extend this work in two directions. First, we plan to evaluate our techniques with other content-extraction algorithms for other media types such as image, voice and face matching functions. Second, we intend to investigate the extension of our approach in order to combine it with other clustering (hierarchical) and indexing methods.

References

[1] *MuscleFish Audio Information Retrieval (AIR) Datablade Module User'sGuide, Version 1.4*. Informix Software, Inc., Menlo Park, California, July 1997.

[2] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, June 1998.

[3] J.R. Bach, C. Fuller, A. Gupta, A. Hampapur, B. Horowitz, R. Humphrey, F. Jain, and C. Shu. The virage image search engine: An open frame work for image management. In *SPIE Storage and Retrieavl for Image and Video Databases IV*, volume SPIE 2670, pages 76–87, Feb 1996.

[4] G. Barish, C. Knoblock, Y. Chen, S. Minton, A. Philpot, and C. Shahabi. Theaterloc: A case study in building an information integration application. To appear in IJCAI Workshop on Intelligent Information Integration.

[5] N. Beckmann, H.-P.Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD International Conference on the Management of Data*, pages 322–331, May 1990.

[6] T. Blum, D. Keislar, J. Wheaton, and E. Wold. *Audio Analysis for Content-Based Retrieval.* MuscleFish, 1996.

[7] M. G. Christel, T. Kanade, M. Mauldin, R. Reddy, M. Sirbu, S. M. Stevens, and H. D. Wactlar. Informedia digital video library. *CACM*, 38(4):57–58, 1995.

[8] N. Dimitrova and F. Gholshani. Rx for semantic video database retrieval. *ACM Multimedia*, pages 219–226, 1994.

[9] N. Dimitrova and F. Gholshani. Video and image content representation and retrieval. In *Handbook of Multimedia Information Management*, 1997.

[10] R. Duda and P. Hart. *Pattern Classification and Scene Analysis.* New York: John Wiley ans Sons, 1973.

[11] U. Fayyad, D. Haussler, and P. Stolorz. Mining Science Data. *Communications of the ACM*, 39(11), 1996.

[12] D. Fisher. Knowledge Acquision via Incremental Conceptual Clustering. *Machine Learning*, 2(11):1987, 139-172.

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

[14] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* San Diego, CA: Academic Press, 1990.

[15] V. N. Gudivada and V. Raghavan. Content-based Image Retrieval Systems. *IEEE Computer*, 28:18–22, 1996.

[16] J. Hafner, H. Sawhney, W. Equitz, M. Flickner, and et al. Efficient color histogram indexing for quadratic form distance functions. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 729–736, July 1995.

[17] J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, September 1995.

[18] L. Kaufman and P. Rousseeuw. *Finding Groups in Data.* John Wiley ans Sons, 1989.

[19] K.-I. Lin, H. Jagadish, and C. Faloutsos. The TV-tree - an index structure for high-dimentional data. In *VLDB Journal*, pages 517–542, Oct. 1994.

[20] W. Meng, C. Yu, W. Wang, and N. Rishe. Performance Analysis of Several Algorithms for Processing Joins between Textual Attributes. In *Proc. of the 12th IEEE International Conference on Data Engineering*, pages 636–644, February 1996.

[21] H. Zhang Q. Wei and Y. Zhong. Robust approach to video segmentation using compressed data storage and retrieval for image and video databases. In *SPIE*, pages 448–456, 1997.

[22] C. Shahabi, M. Alshayeji, N. Jiang, and L. Khan. Improving the Performance of Audio-based Similarity Queries with Clustering. In *Acm First International Wprkshop on Multimedia Intelligent Storage and Retrieval Management*, Oct. 1999.

[23] B. Shahraray. Scene change detection and content based sampling of video sequences: Digital video compression, algorithms and technologies. In *SPIE 2419*, Feb. 1995.

[24] Julius T. Tou and Rafael Gonzalez. *Pattern Recognitio Principles*, chapter 3, pages 75–109. Addison-Wesley Publishing Company, 1981.

[25] Asha Vellaikal and C.-C. Jay Kuo. Hierarchical Clustering Techniques for Image Database Organization and Summarization. In *SPIE Multimedia Storage and Archiving Systems III*, pages 68–79, November 1998.

[26] David White and Ramesh Jain. Similarity Indexing with the SS-tree. In *Proc. of the 12th IEEE International Conference on Data Engineering*, pages 516–523, February 1996.

[27] H. Zhang, C. Y. Low, S. W. Smoliar, and D. Zhang. Video parsing;retrieval and browsing: An integrated and content-based solution. *ACM Multimedia*, pages 15–24, 1995.