A Meta-Index to Integrate Specific Indexes: Application to Multimedia*

Ilaria Bartolini†, Paolo Ciaccia
DEIS - University of Bologna, Italy
{*ibartolini, pciaccia*}@deis.unibo.it

Lei Chen ‡
Hong Kong University of Science and Technology, China
leichen@cs.ust.hk

Vincent Oria
New Jersey Institute of Technology, USA
vincent.oria@njit.edu

Abstract

Although applications such as news, e-learning, e-commerce require multimedia data of different types to co-exist, data for each media type are often managed separately because technically they do not have much in common. The common approach for providing the user with an integrated multimedia database is to build an application layer that presents a unified interface and deals with the right mono-medium sub-system depending on the user request. The problem is that this approach does not fully integrate the underlying systems. Another solution is to push the integration into the query processor. This second approach necessitates to have access to the query processor which is not always possible especially if the multimedia system is using a commercial DBMS underneath. In this paper, we propose a meta-index that hosts different specific indexes and integrates the sub-query results to be returned to the user.

1. Introduction

Database Management Systems (DBMS) offer some functionalities (transaction management, declarative query languages, recovery, etc.) that are necessary for novel applications (e.g., multimedia and bio-informatics). Although current DBMSs offer solutions to store and manage the complex data that the novel applications manipulate, they cannot be used effectively for such applications. Even though more suitable indexes exist for the specific application, current DBMSs do not allow specific indexes to be "plugged-in". The reason is that indexes are to be used by the query processor as access methods and a new index cannot be used if the query processor is not aware of its existence. Most of the time, the query processor is not accessible to the users. This also means that a new index will not be recognized and used by the query processor.

For classical data, the B^+-tree has been widely accepted as the index structure and offered by Relational DBMSs. Such a standard does not exist for multidimensional data as multidimensional indexes are very specialized [9]. For example, range queries work better with an index of the R-tree family [10], whereas similarity search applications perform better with an index structure such as the SS-tree [15], the SR-tree [11] or the M-tree [6]. Further, different multidimensional indexes can be used for the same application. The endless increasing number of multidimensional index structures leave a lot of choices but pushes the choice of the index to the application designer.

In this paper we propose a meta-index that allows (1) a database system to integrate any user-defined index and (2) an application developer to pick the appropriate indexes for his/her applications. We define the operators needed at the meta-index level to perform the integration of partial results based on partial orders. Then we show how the meta-index can be used together with a unified semantic index to create a full multimedia database system. The rest of the paper is organized as follows: Section 2 depicts the meta-index, Section 3 presents the semantic model for media object annotation, Section 4 presents the experimental results and Section 5 concludes.

2. The Meta-Index

We propose the meta-index as an interface between the query processor and the indexes (see Figure 1). The idea is to have the meta-index implemented in a database system

*This work is partially supported by the WISDOM MIUR Project
†Part of the work was performed when the author was visiting NJIT
‡Work performed when the author was affiliated with NJIT

in order to facilitate the integration of new indexes into the system. The meta-index is viewed by the query processor as a single index that is responsible of certain types of queries. As such, the meta-index has to be in charge of some of the

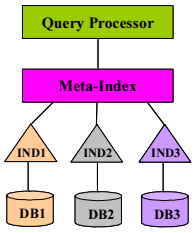


Figure 1. Query Processor, Meta-Index and Indexes

tasks typically devoted to the query processor: Decompose queries it receives into sub-queries to be sent to individual indexes; integrate the sub-query results received from individual indexes.

For the meta-index, an index refers to a piece of software that provides sorted access and/or random access to some data as result of a query. In that sense, for the meta-index an index can range from a regular index to a specific database system.

2.1. Query Decomposition

The meta-index input is a well-formed query in disjunctive normal form, $Q = Q_1 \vee \ldots \vee Qn$, where Q_i is a mono-answer space query. We say that a query $Q_i = Q_i^1 \wedge \ldots \wedge Q_i^k$ is a mono-answer space query if each sub-query Q_i^j is a mono-index query (i.e., results are obtained from a single index) that expresses parts of the conditions of a query (Q_i) in conjunctive normal form. By well-formed we refer to the classical correctness rules for queries: A query has a single free variable (a query returns one type of objects as results) and all the variables are range-restricted. For example, the query "find images similar in color and texture to a given image" is a mono-answer space query and each sub-query (on color and texture, respectively) is a mono-index query assuming that the features are indexed separately. The integration producing the final result is performed within a single answer space. The term answer space comes from of a graphical interpretation of sub-result integration where each sub-query result is represented by an axis in a vector space.

When the meta-index receives a query Q from the query processor, the idea is to decompose this query into $\{Q_1, \ldots, Q_n\}$ so that each sub-query Q_i could be answered through one answer space. That means that each of the corresponding sub-queries Q_i^k uses one or more predicates defined on the same objects. Hence the decomposition consists in: Grouping sub-queries with conditions that involve predicates on the same index; decomposing sub-queries with complex conditions that involve predicates on different indexes into smaller sub-queries with conditions on single indexes.

2.2. The Meta-Index as an Interface

Normally, any index can be added to the meta-index by the application developer. When an index is plugged into the meta-index, the application developer should also specify the types of queries the index should be in charge of. Basically, the query processor needs to be notified on the type of queries that should be passed to the meta-index and the meta-index should know the type of queries to be passed to each index.

Since the queries are in disjunctive normal form, the meta-index needs to know the type of atomic formulas to be sent to individual indexes. Indexes are used as access methods to select a subset of a set (table in the relational world) that satisfy a certain condition defined on at least one of the features of the objects. The conditions are defined using some predicates (i.e. $=, <, >$ in relational). These conditions are used to define atomic formulas: For example, $t.A = constant$, $t.A = s.B$, $t.A < s.B$ are atomic formulas on attributes A and B of tuples $t \in R$ and $s \in S$. If there is an index IND_i defined on the attribute A of R that works for the predicates $<, >, =$, then atomic formulas of the form $t.A < constant$, $t.A > constant$ and $t.A = contant$ should be sent to IND_i for $t \in R$. So when parsing a query, the meta-index should look for $t \in R$ and at least one of the following atomic formulas: $t.A < constant$, $t.A > constant$ and $t.A = constant$ in order to determine if part of the query should be sent to the index IND_i.

When the meta-index gets the lists of atomic formulas that each index is in charge of, it compiles a complete list of atomic formulas that it can handle and notifies the query processor. The meta-index is between the query processor and the indexes. It is responsible for decomposing queries received into mono-index sub-queries, integrating the sub-query results received and sending the result back to the query processor.

2.3. Operators for Sub-Query Result Integration

For the meta index we define three operators: The join with order, the difference with order and the union with order as extensions respectively of the join, the difference and the union relational operators. These operators are defined in terms of partial orders because we have to be able to reason about the queries and their results at the meta-

index level without knowing the metric used to define them. As in relational databases, the conjunctive connectives are mapped to joins with order, a negation is a difference with order and a disjunctive connective is a union with order.

2.3.1 Join with Order

The result of a query posed against an index is a collection R of objects (e.g., "list images taken by Ilaria") or a ranked list (e.g., "list images similar to the query image"). Although ranked lists are usually based on a metric, for the sake of generality we represent a ranked list as a collection R of objects with a partial order $\prec$, i.e., $[R, \prec]$, since total orders are particular cases of partial orders. Note that $[R, \prec]$ actually is a graph, where nodes are elements of R and an arc (x, y), with $x, y \in R$, means that "x is better than y".

Complex queries can involve the integration of the results of two ranked lists from different indexes. For example, "images similar to a query image on texture and on color", assuming that colors and textures are indexed separately. The cases where the integration is about unordered collections is similar to the relational case. The novelty is introduced by the orders and here we will focus on these.

Definition 2.1 (Join with Order) *Given two collections R and S of objects of the same type, the condition of the partial order join is that the objects in both sets be identical and the join with order between two collections R and S, $R \bowtie_{\prec} S$, is defined as follows:*

***Case 1**: R and S are sets: $R \bowtie_{\prec} S = R \bowtie S = R \cap S$ (recall that the join condition is that the objects are identical).*

***Case 2**: One of the collections (let us say R with the partial order $\prec_r$) is a ranked list: $R \bowtie_{\prec} S = [R, \prec_r] \cap S = [R \cap S, \prec]$, where for $x \in R - S$ and $y \in R$, $\prec = \prec_r - \{(x, y), (y, x)\}$.*

***Case 3**: Both R and S are ranked lists ($[R, \prec_r], [S, \prec_s]$): $R \bowtie_{\prec} S = [R, \prec_r] \bowtie_{\prec} [S, \prec_s] = [R \cup S, \prec]$, where $\prec$ defines a global partial order among the objects in R and S (see discussion).*

We will discuss only Cases 2 and 3 as Case 1 is identical to the join in the relational model. An example of query of Case 2 is Q: "list images taken by Ilaria that look like the image i". Given a collection S of images taken by Ilaria and a ranked list $[R, \prec_r]$ of images similar to image i, the aim is to select the objects of R that also belong to S and define a new partial order on the qualifying objects. This is derived from $\prec$ by simply removing from it all the pairs (x, y) where x is not part of the result, $x \in R - S$ (i.e., all images not taken by Ilaria).

Given at least 2 sub-query results with partial orders, the idea in Case 3 is to compute a set with only one partial order. Each sub-query deals with one index and the final result is computed starting from sub-query partial results. The question is which order to use for the final result. We discuss the solutions in Section 2.4.

The model of queries we consider includes the standard one, where the user is interested in obtaining the top-k results, the major difference being, of course, the criterion according to which objects are ranked. To this end, we rely on the well-defined semantics of the $Best$ [14] and *Winnow* [5] operators, recently proposed in the context of relational DBs. The Best operator, $Best(R)$, returns all the objects o in R such that there is no object in R better than o according to $\prec$. Ranking can be easily obtained by recursively applying the Best operator to the remaining objects (i.e., those not in $R - Best(R)$, and so on). This leads to a *layered* view of the search space where all the objects in one layer are "indifferent".

2.3.2 Difference with Order

This is equivalent to integrating the results of sub-queries with negation.

Definition 2.2 (Difference with Order) *Given two collections R and S of objects of the same type, the difference with order between two collections R and S, $R -_{\prec} S$, is defined as follows:*

***Case 1**: R and S are sets: $R -_{\prec} S = R - S$.*

***Case 2 (a)**: R is a set and S is a ranked list $[S, \prec_s]$: $R -_{\prec} [S, \prec_s] = R - S$.*

***Case 2 (b)**: R is a ranked list $[R, \prec_r]$ and S is a set: $[R, \prec_r] - S = [R - S, \prec]$ where for $x \in R \cap S$ and $y \in R$, $\prec = \prec_r - \{(x, y), (y, x)\}$.*

***Case 3**: Both R and S are ranked lists ($[R, \prec_r], [S, \prec_s]$). Since the difference of two partial orders is not a partial order anymore (the transitivity property of partial orders cannot be guaranteed), this case is not considered.*

Let us consider the following two queries to illustrate the definition: Q_1: "list images taken by Ilaria that do not look like the image i" and Q_2: "list images that look like the image i that are not taken by Ilaria". Again let us assume that S is the collection of images taken by Ilaria and $[R, \prec_r]$ the ranked list of images that are similar to the image i. Query Q_1 corresponds to Case 2 (a), where the result should be a set of image taken by Ilaria, that is, $S - R$, thus ignoring the partial order associated to R. This makes sense if R contains a top-k-like answer where not all the objects in the database are in the result, otherwise the result of the query will always be empty. Query Q_2 corresponds to Case 2 (b), where the result should consist of images of R that do not belong to S, with a new partial order that involves only the objects that are in R but not in S. This is obtained from $\prec$ by removing from it all the pairs (x, y) where $x \in R \cap S$ (i.e., images taken by Ilaria).

2.3.3 Union with Order

Definition 2.3 (Union with Order) *Given two collections R and S of objects of the same type, the union with order between two collections R and S $R \cup_{\prec} S$ is defined as follows:*

***Case 1**: R and S are sets: $R \cup_{\prec} S = R \cup S$.*

***Case 2**: R is a set and S is a ranked list $[S, \prec_s]$: $R \cup_{\prec} [S, \prec_s] = [R \cup S, \prec_s]$.*

***Case 3**: Both R and S are ranked lists ($[R, \prec_r]$ and $[S, \prec_s]$). Again, taking the union of two partial orders do not yield a partial order. For this reason the union is not performed, but the two ranked lists are returned separately as alternate solutions. Note that this makes sense even when sub-results have different types (e.g., text and image).*

Let us briefly comment on Case 2. Since, by definition, a partial order does not require that all the elements are ordered, when we take the union of a set R and a ranked list $[S, \prec_s]$, we can simply add the objects in $R - S$ without having to change the order of the objects in S.

2.4. Operator Implementation

The implementation of the *difference with order* is not too much different from a set difference with sorted and random access. We will then focus on the *join with order*.

Given a query divided in independent sub-queries, the goal is to provide the user with the set of top results minimizing the number of objects to be accessed for it. More in details, the top-k retrieval paradigm is applied, by splitting the initial complex query into a set of m simpler sub-queries. Each sub-query deals with only some of the query predicates and the final result is computed starting from sub-query partial results. A relevant example of top-k queries are "middleware" queries [7, 8, 3, 2], where the best k objects are retrieved given the (partial) descriptions provided for such objects by m distinct data sources. In this scenario, a first assumption is that each data source is able to return a ranked list of results. More precisely, each returned object comes with an ID that identifies the object in the data source plus a score that numerically quantifies in which measure the object matches the query on that data source (named *partial* score). By means of the $getNext()$ method it is possible to execute a *sorted access* obtaining the best next object together with its partial score.

With regard to the aggregation rule of partial scores, one possibility is to use a *scoring function*, such as the (weighted) *average*, the *maximum* and the *minimum*, that aggregates the m partial scores into a global similarity score. In this case, objects are linearly ordered and only the highest scored ones returned to the user. A more general solution, that contains the scoring function-based approach as special case, consists in adopting *qualitative preferences* (e.g., *Skyline* [1] and *Region-prioritized Skyline* [14]) able to define arbitrary partial orders on the objects. Moreover, with qualitative preferences a more flexible comparison criteria able to take into account all the partial scores is possible. The only requirement of this aggregation modality is thus to define a binary preference relation able to assert when an object is "better" than another one.

Although each of the above integration solutions could be applied to implement our operators, in this work we adopted the iMPO [2] algorithm. The aim of this paper, in fact, is to find a minimum common denominator to the involved indexes, with or without ranking, and to propose an algebra to integrate their results, independently from the specific integration choice; the only requirement being that it just fit our framework.

iMPO is based on a partial order approach which means that the meta-index can integrate the results of the indexes without having to know the metrics they are using. In particular, for the computation of the top-k results we rely on the *BestTop* [2] operator that combines the semantics of the *Best* and top-k operators. BestTop recursively applies Best to the objects by computing the first l layers containing the top-k results.

3. The Meta-Index for Multimedia

Although users want to be able to manipulate in the same application any type of medium and multimedia data (e.g., images, audio, documents and video), data of different media types are totally different: They have different representation schemes, different features, different possible queries and they need different tools to render the data. The MPEG-7 standard [4] gives an exhaustive description of the features for each media type. But different media objects can share the same semantics and a semantic index common to all the media type can help integrate the different media types and build a full multimedia application. Figure 2 depicts the functional architecture of the meta-index for a multimedia application in which the semantic index plays a central role. Although the semantic index is put at the same level as the other indexes, it references object in the other indexes.

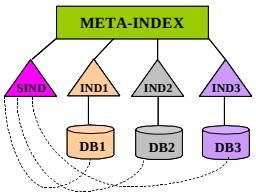


Figure 2. Semantic Integration of Multimedia Data with the Meta-Index

To that end, we propose a simple multimedia type system that follows the common multimedia system organization where each media type is handled by a specialized subsystem. As also shown in Figure 3, the type system should be rooted with *multimedia* as the root type. The rooted type system is important as it allows the semantic index to refer to only one type of objects (e.g., *multimedia*). The multimedia type defines two types of features, synthetic metadata (low-level features extracted from raw multimedia data) and semantic information (salient objects). In Figure 2, $SIND$ is the semantic index shared by all the media types, $IND1$ can be an index on image visual features, $IND2$ can be an index on video, and $IND3$ an index on text. The generic media type with synthetic and semantic information is the minimum required as the integration is based on the common semantics. The goal is not to discuss how to obtain the semantic information but how to use this information in a meta-index to glue data from different media types.

3.1. Media Content Description Model

The description of media objects mainly obeys two structures. The first one is the media class hierarchy and the second one is the media object aggregation hierarchy:

Media Class Hierarchy: The media class structure hierarchy is based on the "IS-A" relationship and defines the properties for each class. We distinguish the media class hierarchy used to group related media objects and the media content hierarchy that defines the descriptors for each media class. Note that the two hierarchies can be combined so that every media object comes with its content descriptors.

Media Object Aggregation Hierarchy: Every media object has an inherent structure. For example, the common hierarchy admitted for a video is: A video is composed of scenes, scenes are composed of shots, shots are composed of key frames (images). On the other hand, the structure of an XML document is given by a DTD or a XML schema.

The media class hierarchy defines the necessary descriptors for each media object depending of its class. It is application dependent so has to be provided when the application is being developed. We capture the user-defined media class hierarchy by attaching it to the right super-class in our pre-defined rooted type system as shown in Figure 3. The

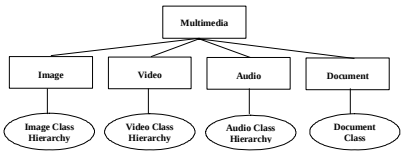


Figure 3. The Multimedia Type System

media object aggregation hierarchy defines different granularity levels for each media object. These granularity levels are important in the content description of the media object. The media class hierarchy and the media object aggregation hierarchy form the *media object metadata scheme* and can be defined using an object-relational model or MPEG-7.

3.2. Media Object Tree

Given a media object metadata scheme and a media object, we can construct the *media object tree* that reflects the composition of the media object and records the descriptors. Figure 4, for example, gives a media tree of a video composed of 2 scenes, 5 shots, and 7 key frames.

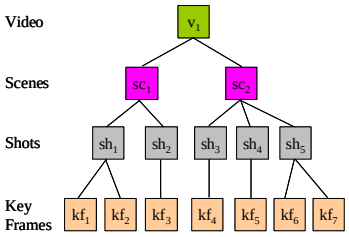


Figure 4. The Media Tree of a Video

The media object trees are media objects together with descriptors. They can be stored as MPEG-7 documents and queried using an XML query language like XQL [16]. The query processing will be more effective with an index. We used the MBM indexing method [12] that extends the BUS [13] method to index the media object trees. Each element of the media object is assigned a Unique element IDentifier (UID) using a left-to-right, top-to-bottom traversal that can be used to calculate the parent UID. The BUS also introduced the concept of the General element Identifier (GID) by extending the UID concept to include (1) Document Number, (2) the UID of the element, (3) the level of the element in the tree and (4) the element type number. The central concept of the BUS method that the MBM method inherited is that indexing information (the GID, term frequency, etc.) are maintained only for the leaf nodes of the media tree (e.g. keyframe for a video). We extended the GID to include the media type number (1:image, 2:video, 3:audio, and 4:document) stored using 2 bits. The media type information helps select the GID of a given media type as all the media types share the same semantic index.

4. Experiments

In the experimental section we design several scenarios to show how the meta-index interacts with the query processor and quantify, by means of a set of preliminary results, the improvement obtained by using the meta-index in term

of both effectiveness and efficiency. All programs are written in *Java* programming language (J2SE, v. 1.5) and experiments are run on a Sun-Blade-1000 workstation under Solaris 2.8 with 1GB of memory. Results are obtained on a simple database which extracts video clips and key frames from two movies[1]. In details, the dataset contains 20 video clips, 230 key frames (images) extracted from the video clips, and 23 salient objects (for semantic queries). From each image, we extracted a 32-bins color histogram represented in the HSV color space and a 60-bin texture vector using Gabor filters. To compare both color and texture feature vectors we used the Euclidean distance.

The most typical queries that we want to test are mono-type (Q_1) and mixed-type media (Q_2) queries defined as following:

Q_1: $\{i$: Image $\mid$ $i \in$ Images, $\exists A \in$ Objects, $\exists B \in$ Objects, i contains $A \wedge$ i contains $B \wedge$ i.color similar query.color $\wedge$ i.texture similar query.texture $\wedge$ A besides $B\}$;

Q_2: $\{i$: MM $\mid$ $i \in$ Images $\vee$ $i \in$ KFs $\exists$ $A \in$ Objects, $\exists$ $B \in$ Objects, i contains $A \wedge$ i contains $B \wedge$ i.color similar query.color $\wedge$ i.texture similar query.texture $\wedge$ A besides $B\}$.

In these formulas, *contains*, *similar* and *besides* are predicates, *MM* is a root type for multimedia, *Images* is a set of images, *KF*s is a set of key frames in videos and *Objects* is a set of salient objects. Finally, *query* is the example query image.

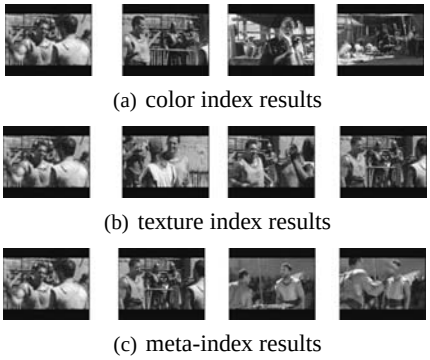
To deal with this type of queries, we implemented the join with order operator (see Section 2.3.1). In particular, we applied join with order (Case 3), for the integration of low-level features results, whereas for the integration with salient objects-based results we adopted join with order (Case 2). We indexed both feature vectors for color and texture using an index structure for high-dimensional nearest neighbor queries (i.e., the M-tree [6]), whereas for salient objects we adopted MBM [12].

All results we present are averaged over a sample of 50 randomly-chosen query images containig two salient objects.

We test the efficacy of meta-index using mono-type (Q_1) and mixed-type media (Q_2) queries. To measure the effectiveness of our solution we consider the classical *precision* metric, i.e., the percentage of relevant images found by a query (in our experiments we set $k = 20$), averaged over the query workload.

For each query Q_1 the meta-index first decomposes the query into mono-type sub-queries and makes a search among registered indexes. Then, each mono-type sub-query is sent to the corresponding registered indexes, i.e., the color index, the texture index, and the salient object index. Results returned from low level image feature indexes (e.g., color and texture) are integrated by the meta-index using iMPO algorithm [2] (this correspond to Case 3). The set of images returned by salient-object index are combined with color-texture integrated results by taking their intersection (Case 2). As for queries Q_2, the meta-index preforms similar steps. This depends on the fact that a query Q_2 is composed by several queries Q_1. Further, in this case, in addition to images, video clips where images are extracted from are also returned.

We compared precision results obtained when the meta-index was switched on with those obtained without meta-index. We start by showing a visual example (depicted in Figure 5), where the top 3 results for the same query (the image on the left most column) are depicted when using a single index on color (a), a single index on texture (b), and the meta-index (c).



(a) color index results

(b) texture index results

(c) meta-index results

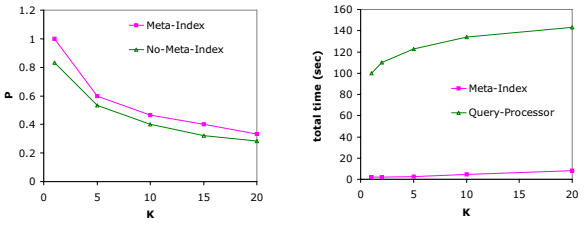
Figure 5. Visual Results using Color Index (a), Texture Index (b) and Meta-Index (c)

By Figure 5, we can find out that with the meta-index, we can achieve more effective results: Images in (c) are much closer to the query in terms of both low level features and semantics than those obtained only using single indexes. The reason being that the meta-index gets the best from low-level features results (i.e., color and texture) and semantic results (i.e., salient objects).

The trend in visual example is generalized by Figure 6 (a), where averaged precision values for the query workload are reported. In particular, we use the term No-Meta-Index to refer to the average precision obtained using single indexes, whereas with Meta-Index results obtained by the meta-index. As we can observe from the graph, Meta-Index outperforms No-Meta-Index for all value of k with a final average improvement of 19.44% (for $k = 15$) and 15% (for $k = 20$), respectively.

We report the efficiency test of using meta-index in answering queries. As for the measure, we consider the running time needed to answer a query based on low-level features (i.e., color and texture) and salient objects with re-

[1] "Gladiator", 2000 and "Life is Beautiful", 2000.



(a) (b)

Figure 6. Precision for Meta-Index and No-Meta-Index vs Number of Retrieved Images k (a) and Running Time Comparison Using Meta-Index and Query Processor (b)

spect to the number of retrieved object k (in our experiments we vary k between 1 and 20), averaged over the randomly picked up queries. For this experiment, we only use mono-type queries (Q_1). As shown in the previous section, queries Q_1 are the bases for mixed-type media queries Q_2. In particular, we assume that not all query features have supporting indexes (i.e., indexes for textures do not exist). Under this assumption, given a query Q_1, the query processor decomposes it into a set of mono-type sub-queries and keeps the sub-queries whose specified features do not have supporting indexes (the query processor can not find the registered index in the interface to match the sub-query) and sends sub-queries with supporting indexes to the meta-index. Results returned from the indexes and the query processor are then integrated by the query processor. Performance is compared between queries using the meta-index and the query processor only. For the queries answered by the meta-index, the execution time can be saved by integrating the results returned by the indexes first, once the results from the query processor are returned, the final results can be intergraded. For the queries that are answered by the query processor directly, the results can only be integrated when all results are returned.

Figure 6 (b) shows average running time results. In particular, we observe how Meta-Index, by using indexes on salient objects (MBM) and low level features (M-tree), is able to achieve better performance than Query-Processor with a final overage improvement of 94.47%. This depends on the correct selection of indexes and the early integration of ranked lists alloweb by the meta-index.

5. Conclusions

In this paper we have proposed a meta-index to integrate specific indexes in a multimedia DBMS. We have defined three operators for performing integration of ranked lists, namely, join with order, difference with order and union with order. These operators are defined in terms of partial orders, which allows an integration of sub-results without knowing the specific metrics used by the underlying indexes to order them. The experimental results confirm the efficacy and efficiency of the meta-index in answering top-k queries.

References

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of the 17th International Conference on Data Engineering (ICDE'01)*, 2001.

[2] I. Bartolini, P.Ciaccia, V. Oria, and M. T. Özsu. Flexible integration of multimedia sub-queries with qualitative preferences. *Multimedia Tools and Applications*, 2006. To appear.

[3] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *Proc. of the 18th International Conference on Data Engineering (ICDE'02)*, 2002.

[4] S.-F. Chang, T. Sikora, and A. Puri. Overview of the MPEG-7 standard. *IEEE Transactions on Circuit and Systems for Video Technology*, 11(6), 2001.

[5] J. Chomicki. Querying with intrinsic preferences. In *Proc. of the 6th International Conference on Extending Database Technology (EDBT02)*, 2002.

[6] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, 1997.

[7] R .Fagin. Combining fuzzy information from multiple systems. In *Proc. of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1996.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2001.

[9] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), 1998.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM International Conference on Management of Data*, 1984.

[11] N. Katayama and S. Satoh. The SR-tree: An index structure for high dimensional nearest neighbor queries. In *Proc. of ACM SIGMOD International Conference on Management of Data*, 1997.

[12] V. Oria, A. Shah, and S. Sowell. Indexing XML documents: Improving the BUS method. In *Proc. of 7th Intl Workshop on Multimedia Information Systems (MIS'01)*, 2001.

[13] D. Shin, H. Jang, and H. Jin. BUS: An effective indexing and retrieval scheme in structured documents. In *Proc. of the International Conference on Digital Library (DL'98)*, 1998.

[14] R. Torlone and P. Ciaccia. Finding the best when it's a matter of preference. In *Proc of Workshop on Recommendation and Personalization in eCommerce (AH'02)*, 2002.

[15] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of the 12th International Conference on Data Engineering (ICDE'96)*, 1996.

[16] R. K. Wong. The extended XQL for querying and updating large XML databases. In *Proc. of the 2001 ACM Symposium on Document Engineering*, 2001.