# An Efficient Query Optimization Strategy for Spatio-Temporal Queries in Video Databases *

*Gülay Ünel, Mehmet Emin Dönderler, Özgür Ulusoy and Uğur Güdükbay*

Department of Computer Engineering, Bilkent University

Bilkent, 06533 Ankara, Turkey

e-mail: {gunel, mdonder, oulusoy, gudukbay}@cs.bilkent.edu.tr

**Abstract**

The interest for multimedia database management systems has grown rapidly due to the need for the storage of huge volumes of multimedia data in computer systems. An important building block of a multimedia database system is the query processor, and a query optimizer embedded to the query processor is needed to answer user queries efficiently. Query optimization problem has been widely studied for conventional database systems, however it is a new research area for multimedia database systems. Due to the differences in query processing strategies, query optimization techniques used in multimedia database systems are different from those used in traditional databases. In this paper, a query optimization strategy is proposed for processing spatio-temporal queries in video database systems. The proposed strategy includes reordering algorithms to be applied on query execution tree. The performance results obtained by testing the reordering algorithms on different query sets are also presented.

**Keywords:** video databases, query optimization, query tree, querying of video data

## 1 Introduction

The interest for Multimedia Database Management Systems (MDBMSs) has grown rapidly with the advances in computer technology. The research on content-based image retrieval by low-level features (color, shape and texture) and keywords (Chang and Fu, 1980; Flickner et al., 1995) has progressed in time towards video databases dealing with spatio-temporal and semantic features of video data. Some video database systems such as VideoQ (Chang et al., 1997),

*This work is supported by the Scientific and Research Council of Turkey (TÜBİTAK) under Project Code 199E025.

KMED (Chu et al., 1995), QBIC (Flickner et al., 1995) and OVID (Oomoto and Tanaka, 1993) were implemented. Querying video objects by motion properties has also been studied (Nabil et al., 2001; Guting et al., 2000; Li et al., 1997; Sistla et al., 1997).

Building blocks for MDBMSs are multimedia data model, multimedia storage management, query interface, and query processing and retrieval. Data models used in MDBMSs are different from those used in conventional DBMSs; therefore, new modeling techniques are required to represent the semantics of multimedia data. Besides, a multimedia storage manager is needed and storage devices capable of storing large volumes of data must be supported to achieve better performance. Query interface in a multimedia database system must enable the user to construct well-defined queries easily. Query processing and retrieval is also important since providing powerful querying facilities on multimedia data is a very crucial issue. The conventional query paradigm of traditional database systems only deals with exact queries on conventional types of data but querying multimedia databases requires additional techniques to support multimedia data types, such as image, audio and video. Extensions to the conventional query languages are required to deal with specific requirements of multimedia data. In addition, different query optimization techniques are also needed.

This paper is concerned with the spatio-temporal queries in video databases. We can consider video data as a set of frames, containing a set of objects in each frame. Objects of each frame have some spatial relationships and they change their locations and their relative positions with respect to each other in time. Because of this, spatial and temporal relationships in a video should not be considered separately; instead, spatio-temporal relationships need to be taken care of together. Spatio-temporal relationships constitute the content of video data and they are used to support content-based retrieval of data in multimedia databases. Content-based retrieval using spatio-temporal relationships is one of the most important differences between multimedia and traditional databases.

In our work, we focus on optimization of queries that involve spatio-temporal relationships in video databases. The query optimization module of a database system is one of the most important parts in determining the performance of the system. The input to this module is some internal representation of a query given by the user. This representation is the query tree in our case. The aim of query optimization is to select the most efficient strategy to access the relevant data and answer the query. Let $S$ be the set of all possible strategies (query trees) that can be used to answer a given query. Each member $s$ of $S$ has a cost *c(s)*. The goal of an optimization algorithm is to find a member of $S$ that has the minimum cost. In this paper, we propose an efficient query optimization strategy for spatio-temporal queries in video databases. Our work concentrates on reordering query trees of spatio-temporal queries in a video database system to achieve the minimum cost. We propose algorithms used for reordering query trees. The basic idea with the optimization algorithms is to change the processing order of subqueries contained in the query tree in order to execute the parts that are more selective (i.e., result in fewer frames and/or objects) first. Two types of reorderings are applied on query trees to achieve more efficient processing of queries: 1)*internal node reordering*, which reconstructs the query tree by reordering the children of internal nodes, and 2)*leaf node reordering*, which restructures the query contents of the leaf nodes of the query tree. The query optimization algorithms have been implemented as a part of the query processor of a video database management system, BilVideo (Dönderler et al., 2000, 2002a, 2002b) and tested using sample videos.

The remainder of the paper is organized as follows. In Section 2, related work on multimedia query optimization is discussed. The video database system, into which query optimization module is integrated, is described in Section 3. In Section 4, our query optimization algorithms are presented. Performance results are discussed in Section 5. Conclusions and future research directions are given in Section 6.

## 2 Related Work

Basic principles of query optimization in database systems are explained in (Jarke and Koch, 1984). In their paper, a wide variety of approaches are proposed that include logic-based and semantic transformations, fast implementation of basic operations, and combinatorial or heuristic algorithms for generating alternative access plans and choosing among them. Nonstandard query optimization issues are also discussed in the paper. According to Jarke and Koch, the goals of query transformation are *standardization*, *simplification*, and *amelioration*. The transformation rules for the general query expressions referenced in the paper are also valid for our query expressions.

Chaudhuri (1998) focuses primarily on the optimization of SQL queries in relational database systems. The paper discusses the System-R optimization framework, search space that is considered by optimizers, cost estimation and enumeration of the search space. The basic cost estimation framework in System-R uses statistical summaries of data that have been stored. The idea of collecting statistical summaries for cost estimation is also used in our query optimization strategy.

Garofalakis (1998) studied query scheduling and optimization in parallel and multimedia databases. He developed a multi-dimensional framework and provably near-optimal algorithms for scheduling both time-shared and space-shared resources in hierarchical and shared-nothing architectures. Garofalakis elaborates on the areas of resource scheduling for composite multimedia objects, on-line admission control for multimedia databases, and scheduling support for periodic models of user service. Our optimization method, on the other hand, is based on finding an optimal query plan rather than using resource scheduling on the architecture that stores the database.

Atnafu et al. (2001) presented similarity-based operators and query optimization for multimedia database systems. They focused on the management of content-based image databases. Currently available content-based image retrieval systems commonly search for the most similar images from a set of images for a given single query image or a feature vector representation of an image. Atnafu et al. introduced the most needed similarity-based operations, studied their properties, formalized the use of them, and used these as a basis for a similarity-based query optimization for image database systems. Their paper defines the similarity-based selection and similarity-based join operations. Similarity-based query optimization is based on the algebraic rules on these similarity-based operations. Our method also uses the algebraic rules on relational operators to reorder the query tree and find an optimal query execution plan. However, we work on video databases rather than image databases. Our query language queries videos, segments of videos or values of variables for given spatio-temporal relations on objects, using fact-bases created for each video in the database, which is significantly different from the query language of a content-based image retrieval system. Hence, we do not use similarity-based operations.

The most relevant work to ours on query optimization in multimedia databases is by Soffer and Samet (1999), which presents optimization methods for processing of pictorial queries specified by pictorial query trees. The optimization strategy proposed in their work for computing the result of the pictorial query tree uses the method of changing the order of processing individual query images in order to execute the parts that are more selective. The selectivity of a pictorial query is based on *matching selectivity*, *contextual selectivity*, and *spatial selectivity.* Matching and contextual selectivity are computed using the statistics stored as histograms in the database that indicate the distribution of classifications and certainty levels in the images. These histograms are constructed when populating the database. Selectivity of an individual pictorial query (leaf) is computed by combining these three selectivity factors. The query language used in their system has different characteristics from the query language we use. Their query language includes only spatial relations in the pictorial query tree and they reorder the tree according to the statistics stored for these spatial relations. Our query language has more complex features, enabling the user to query spatio-temporal relations that will be described in the next section. In the query optimization module of our system, fact base statistics are used to reorder spatial relations. In addition to this, reordering of internal nodes that contain operators, is also provided.

Mahalingam and Candan (2001) propose techniques for performing query optimization in different types of databases, such as multimedia and Web databases, which rely on top-k predicates. Top-k predicates are the k predicates that return the most relevant portion of all possible results. They propose an optimization model that takes into account different binding patterns associated with query predicates and considers the variations in the query result size, depending on the execution order. Their optimization model assigns a value (to be minimized) to all partial or complete plans in the search space. It also determines the output size of the data stream for every operator and predicate in the plan. Hence, the proposed optimization algorithm tries to find the best plan considering the output size of the data stream for operators and predicates, which is also used in our optimization algorithm. The major difference of their optimization algorithm from ours is that the number of query results can also change depending on the query execution order in their work, whereas it is independent of the query execution order in our work.

## 3 BilVideo: A Video DBMS

In this section, a video database management system, BilVideo (Dönderler et al., 2000, 2002a, 2002b) to which the work in this paper has been integrated, is described. BilVideo is a video database management system that supports spatio-temporal, semantic and low-level (color, shape and texture) queries on video data. A spatio-temporal query may contain any combination of spatial, temporal, object-appearance, external-predicate, trajectory-projection and similarity-based object trajectory conditions. The system handles spatio-temporal queries using a knowledge-base, which consists of a fact base and comprehensive set of rules implemented in Prolog, while utilizing an object-relational database to respond to semantic (keyword, event/activity, and category-based), color, shape and texture video queries.

## 3.1 BilVideo System Architecture

Figure 1 illustrates the overall architecture of BilVideo. The system is built on a client-server architecture and the users access the video database on the Internet through its visual query interface developed as a Java client Applet.

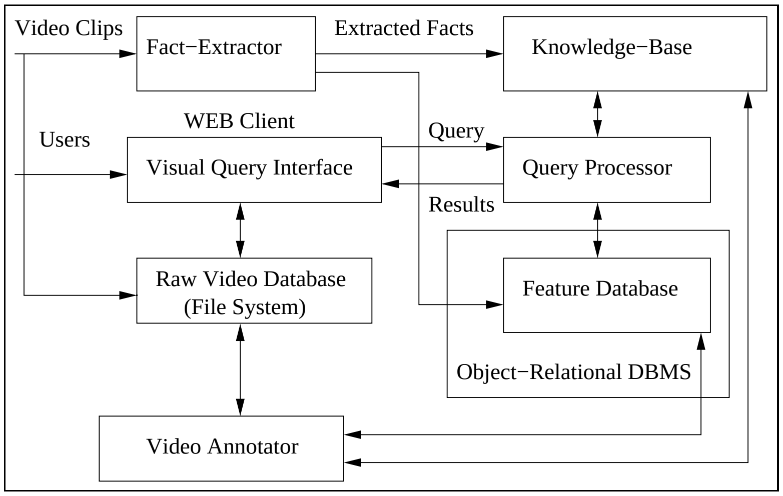


Figure 1: BilVideo database system architecture.

Query processor lies in the heart of the system. It is responsible for answering user queries in a multi-user environment. Query processor communicates with the object-relational database Oracle[2] and the knowledge base. Semantic data is stored in the Oracle database and fact-based meta data is stored in the knowledge base. Video data and raw video data are stored separately. Semantic properties of videos used for keyword, activity/event and category-based queries on video data are stored in the feature database. These features are generated and maintained by a video annotator tool. The knowledge-base is used to answer spatio-temporal queries. The facts-base is generated by the fact-extractor tool.

[2]Oracle is a registered trademark of Oracle Corporation.

## 3.2 BilVideo Query Language

The query language of BilVideo has four basic statements for retrieving information:

```
select {\it video} from {\it all} [where {\it condition}];
select {\it video} from {\it videolist} where {\it condition};
select {\it segment} from {\it range} where {\it condition};
select {\it variable} from {\it range} where {\it condition};
```

The target of a query is specified in `select` clause. A query may return videos (*video*), segments of videos (*segment*), or values of variables (*variable*) with/without segments of videos where the values are obtained. Variables might be used for object identifiers and trajectories. If the target of a query is video (*video*), the users may also specify the maximum number of videos to be returned as a result. The range of a query is specified in the `from` clause. The *range* may be either the entire video collection or a list of specific videos. Query conditions are given in the `where` clause. In our query language, *condition* is defined recursively and it may contain any combination of spatio-temporal conditions. As a consequence of this, the `where` clause can contain spatial conditions, trajectory conditions and the supported operators.

**Supported Operators:** Our query language supports a set of logical and temporal operators to be used in query conditions. Logical operators are *and*, *not*, and *or* while temporal operators are *before*, *during*, *meets*, *overlaps*, *starts*, *finishes*, and their inverse operators, *ibefore*, *iduring*, *imeets*, *ioverlaps*, *istarts*, *ifinishes.* In addition to these, the operators '=' and '!=' are used for assignment and comparison. The query language also has a trajectory-projection operator, *project*, which is used to extract subtrajectories of video objects on a given spatial condition.

Our query language supports spatio-temporal, semantic and low-level queries. Different query types that can be specified by the query language are object queries, spatial queries, similarity-based object-trajectory queries, temporal queries, aggregate queries, low-level queries and semantic queries.

### 3.2.1 Examples

*Query 1:*

```
select X, Y
```

```
from video
where west(X,Y) and disjoint(X,Y)
```

This query searches for objects, X and Y in the video where X is to the west of Y and X is disjoint from Y.

*Query 2:*

```
select segment, X, Y
from video
where (samelevel(X,Y) before disjoint(X,Y)) and
       (infrontof(X,Y) and tr(X, [[west], [1]]))
```

This query searches for objects, X and Y in the video where X is on the same level as Y before X is disjoint from Y, and X is in front of Y, and X has a trajectory to the direction west with displacement 1. The query returns objects X, Y and the identified segments from the video.

## 3.3 Query Processing

Figure 2 illustrates how the query processor communicates with Web clients and the underlying system components to answer user queries. Web clients make a connection request to the query request handler, which creates a process for each request passing a new socket for communication between the process and the Web client. Then, user queries are sent to the processes created by the query request handler. The queries are transformed into SQL-like textual query language expressions before being sent to the server if they are specified visually. After receiving the query from the client, each process calls the query processor with a query string and waits for the query answer. When the query processor returns, the process communicates the answer to the Web client issuing the query and exits. The query processor first groups spatio-temporal, semantic, color, shape and texture query conditions into proper types of sub-queries. Spatio-temporal subqueries are reconstructed as Prolog-type knowledge-base queries. Semantic, color, shape and texture sub-queries are sent as SQL queries to an object relational database. Query processor integrates the intermediate results and returns them to the query request handler, which communicates the final results to Web clients. The details of our query processing system can be found in (Dönderler et al., 2002b).

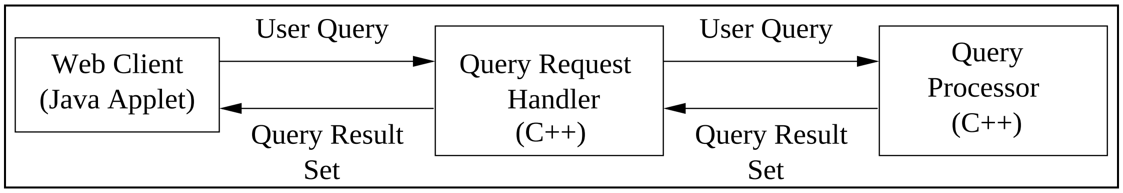


Figure 2: Web client - query processor interaction.

```
InternalNodeReorder(querytree);
ReadStatistics();
LeafNodeReorder(querytree);
```

Figure 3: Query optimization process

## 4 Query Optimization

The aim of the query optimization algorithms designed and implemented for BilVideo is to process more selective subqueries earlier than the others. The algorithms restructure the initial query tree and construct an optimal query tree for that purpose.

The query optimization process implemented during query execution has two basic parts, which are *internal node reordering* and *leaf node reordering*. In addition to these parts, the statistics collected for the video is read from a file before executing the leaf node reordering algorithm (see Figure 3). These statistics are used to determine the selectivities of relations in the condition part of the query. Selectivity of a relation is inversely proportional to the number of facts stored for that relation. Internal node reordering algorithm reorders the children of internal nodes by placing the right child of 'AND' nodes, which are more selective than the left child, to the left of their parents. Leaf node reordering algorithm deals only with the leaf nodes. Every leaf node in the query tree has a content that stores the subquery to be executed. Leaf node reordering algorithm restructures these contents. It uses the subquery trees constructed for each of these contents in the construction of the initial query tree. This algorithm sorts the relations in the contents of the leaf nodes that are connected by 'AND' operators according to their selectivity. More selective operations are executed earlier than the others through the reorderings of the query tree.

## 4.1 Structure of the Query Tree

In our video database model, a query is represented by a query tree. The condition in the `where` clause of the query is kept in this query tree. The condition part can contain spatial relationships. Other functions that can take place in the condition part are object trajectory and project type query functions. Trajectory queries find out the object(s) and/or frame interval(s) of the object(s) having a similar trajectory in a video to a given trajectory. Project queries are used to extract sub-trajectories of video objects on a given spatial condition. The boolean (logical) operators of the query language are *and*, *or*, and *not*. The operators that can be included in a query are categorized into three types:

1. AND: `and`
2. NOT-OR: `not`, `or`
3. TEMPORAL: `before`, `during`, `meets`, `overlaps`, `starts`, `finishes`, and their inverse operators, `ibefore`, `iduring`, `imeets`, `ioverlaps`, `istarts`, `ifinishes`.

The logical operators are categorized into two types as 'AND' and 'NOT-OR' because *not* and *or* operators do not reduce the output size of the operation but *and* operator reduces the output size given any two sets as operands.

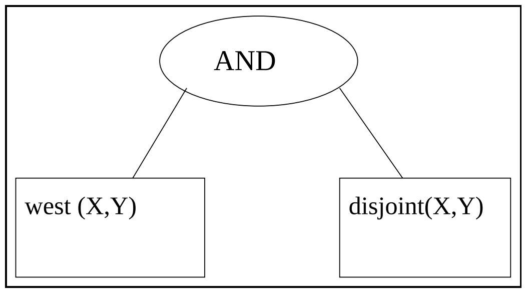
There are two types of nodes in the query tree: internal nodes that contain the operators defined above and leaf nodes that contain spatio-temporal subqueries. These subqueries have three types:

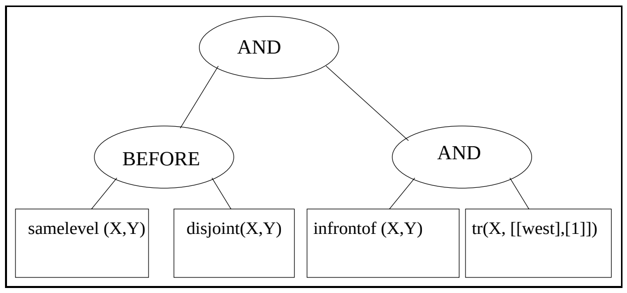
1. Plain Prolog Queries (PPQ): These are spatial subqueries processed by Prolog. They consider the relative positioning of the salient objects with respect to each other. This relative positioning consists of directional, topological and 3-D relations.
2. Trajectory Queries (TRQ): These are object-trajectory subqueries. Trajectory of a salient object is described as a path of vertices corresponding to the locations of the object in different video keyframes. An object-trajectory subquery includes an object name which can be a constant or a variable, a direction list for the trajectory path and the list of displacement values corresponding to each direction given in the direction list. Object-trajectory queries are similarity based; therefore, a similarity value can also be specified.

3. Project Queries (PRQ): The subqueries which contain the trajectory-projection operator, *project* are categorized as a different type.

### 4.1.1 Examples

The query trees for the example queries *Query 1* and *Query 2* given in Section 3.2.1 are shown in Figure 4.





(a) (b)

Figure 4: (a) Query tree for *Query 1* and (b) Query tree for *Query 2* given in Section 3.2.1

## 4.2 Internal Node Reordering Algorithm

In the query tree, the internal nodes are reordered first. Internal node reordering algorithm places the more selective nodes as the left child of their parents, since the left child of a parent is processed first. The proposed algorithm iterates on the query tree and restructures the tree to get the optimal internal-node-structured query tree. The internal node reordering algorithm is given in Figure 5.

The internal node reordering algorithm iterates on the query tree and reorders the children of ‘AND’ typed nodes such that:

- ‘AND’, ‘TEMPORAL’, ‘PPQ’, ‘PRQ’, ‘TRQ’ type child nodes must be on left if the other child is ‘NOT-OR’ type. Since ‘NOT-OR’ type nodes combine results from two different result sets or take the difference of a defined universal set and a given set, they are found to be the least selective compared to the other nodes. They are less selective than ‘PPQ’, ‘PRQ’, ‘TRQ’ type child nodes, because ‘NOT-OR’ type child nodes combine the results of their two subtrees having ‘PPQ’, ‘PRQ’, ‘TRQ’ type nodes as their leaf nodes. They

```
InternalNodeReorder(QueryNode qnode)
 // Process the nodes that have children both on left and right
 if (qnode->Left != NULL and qnode->Right != NULL)
   begin
      type=qnode->Type
      ltype=qnode->Left->Type
      rtype=qnode->Right->Type
      // Reorder the children of ‘AND’ nodes
      if (type==AND)
        begin
          // ‘AND’, ‘TEMPORAL’, ‘PPQ’, ‘PRQ’, ‘TRQ’ type child
          // nodes must be on left if the other child is
          // ‘NOT-OR’ type
          if (ltype==NOT-OR and
             (rtype==AND or rtype==TEMPORAL or rtype==PPQ
                      or rtype==PRQ or rtype==TRQ))
              exchange(qnode->Left, qnode->Right)
          // ‘AND’ type child nodes must be on left
          // if the other child is ‘TEMPORAL’ type
          else if (ltype==TEMPORAL  and rtype==AND)
              exchange(qnode->Left, qnode->Right)
          // ‘PPQ’ type child nodes with zero global variables
          // must be on left if the other child is
          // ‘PRQ’ or ‘TRQ’ type
          else if ((ltype==PRQ or ltype==TRQ) and
                  ((rtype==PPQ) and (gvcount(qnode->Right)==0)))
              exchange(qnode->Left, qnode->Right)
          // ‘PRQ’, ‘TRQ’ type child nodes must be on left if
          // other child is ’PPQ’ type with global variables
          else if (((ltype==PPQ) and (gvcount(qnode->Left)>0))
                      and (rtype==PRQ or rtype==TRQ))
              exchange(qnode->Left, qnode->Right)
          // ‘PRQ’ type child nodes must be on left
          // if the other child is ‘TRQ’ type
          else if (ltype==TRQ and rtype==PRQ)
              exchange(qnode->Left, qnode->Right)
          // ‘TRQ’ type child nodes with zero global
          // variables must be on left if the other
          // child is ‘TRQ’ type with global variables
          else if ((ltype==TRQ) and (gvcount(qnode->Right)>0)
              and (rtype==TRQ) and (gvcount(qnode->Right)==0))
              exchange(qnode->Left, qnode->Right)
        end
   end
 // call the function recursively for left and right subtrees
 InternalNodeReorder(qnode->Left)
 InternalNodeReorder(qnode->Right)
```

Figure 5: Internal node reordering algorithm

are also less selective than 'AND' and 'TEMPORAL' nodes. This is due to the fact that given the same operands, 'AND' and 'TEMPORAL' type operators produce smaller result sets than that of 'NOT-OR' type operators.

- 'AND' type child nodes must be on left if the other child is 'TEMPORAL' type. We argued that AND' and 'TEMPORAL' type operators produce a smaller result set given two operands. We process an 'AND' type node before a 'TEMPORAL' type node because of the fact that 'AND' type nodes are processed faster than the 'TEMPORAL' type nodes since they contain logical operators and our query engine processes logical operators faster than the temporal operators.

- 'PPQ' type child nodes with zero global variables must be on left if the other child is 'PRQ' or 'TRQ' type. This is because of the fact that 'PPQ' type nodes with zero global variables are processed faster and they are more selective than 'PRQ' and 'TRQ' type nodes. 'PPQ' type nodes with zero global variables require a simple search operation on the fact base, however 'PRQ' and 'TRQ' type nodes can require the processing of a project operator or similarity calculation in addition to the same type of seach operation in the fact base. They can also have global variables which can slow down their processing.

- 'PRQ', 'TRQ' type child nodes must be on left if the other child is 'PPQ' type with global variables. This is because of the fact that 'PRQ' and 'TRQ' type nodes are found out to be more selective than 'PPQ' type nodes with global variables. They are found out to be more selective because our sample fact base contains more spatial facts than the trajectory facts. However, this is also the case in most fact bases in real life because spatial facts capture the spatial relations of an object with other objects in all different frames of a video. Therefore, we will usually have more than one spatial relation for an object in the video, while we can have at most one trajectory for an object in the video.

- 'PRQ' type child nodes must be on left if the other child is 'TRQ' type. This is because of the fact that the subquery in the 'PRQ' node can have a variable to be used by the subquery contained in the 'TRQ' node and in this case it is essential to place 'PRQ' type child nodes on left for the correctness of the output of our query processor. We do not check if this is the case or not and put every 'PRQ' type child node on left if the other child is 'TRQ' type, because it does not matter which of the nodes is processed first for the performance, and such a check will induce an execution overhead.

- 'TRQ' type child nodes with zero global variables must be on left if the other child is 'TRQ' type with global variables. This is due to the fact that 'TRQ' type nodes with zero global variables are more selective than 'TRQ' type nodes with global variables. This is because of the fact that we have to search for a single trajectory fact in our fact base in the case of zero global variables, but we have to search for a set of trajectory facts in the case of non-zero global variables.

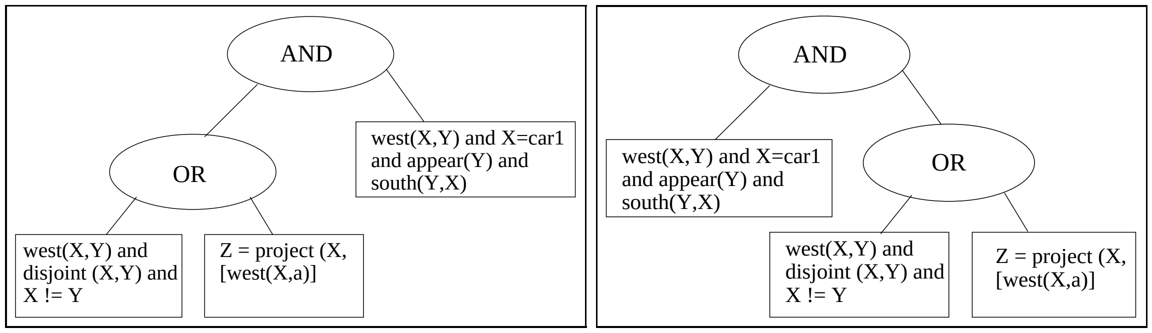
The query tree is restructured using the rules described above. The function *gvcount* in the algorithm (Figure 5) finds out the global variable count of a particular node.

### 4.2.1 Examples

Some query tree examples are given in this part. In each example, the initial query tree and the query tree after internal node reordering are shown.

*Query 1:*

```
select segment, X, Y
from video
where ((west(X,Y) and disjoint(X,Y) and X != Y)
        or Z=project(X, [west(X,a)])) and
        (west(X,Y) and X=car1 and appear(Y) and south(Y,X))
```


AND
OR
west(X,Y) and X=car1 and appear(Y) and south(Y,X)
west(X,Y) and disjoint (X,Y) and X != Y
Z = project (X, [west(X,a)]
AND
west(X,Y) and X=car1 and appear(Y) and south(Y,X)
OR
west(X,Y) and disjoint (X,Y) and X != Y
Z = project (X, [west(X,a)]

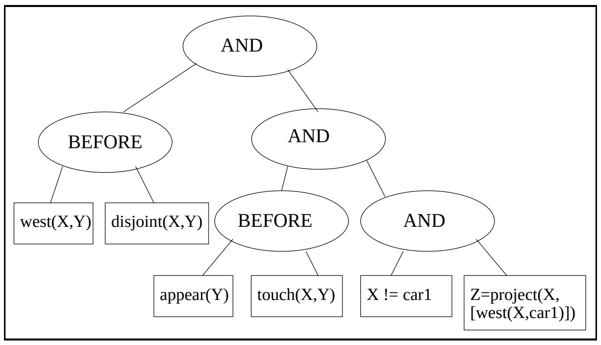
(a) (b)

Figure 6: (a) Initial query tree for *Query 1* and (b) Query tree for *Query 1* after internal node reordering

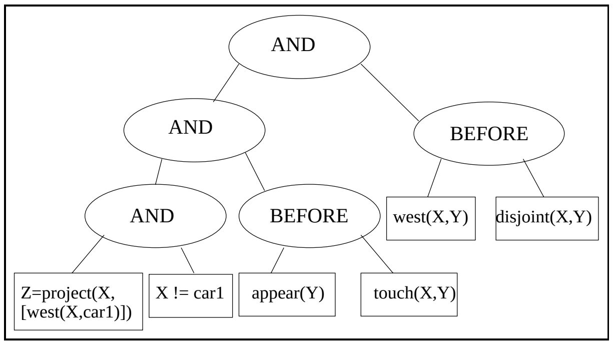
In the query tree of *Query 1*, the children of the root 'AND' node are exchanged since the type of the left child is 'NOT-OR' and the type of the right child is 'PPQ' in the initial query tree (see Figure 6). Our query processor traverses the query tree in postorder, executing each subquery separately and performing interval processing so as to obtain the final set of results. Hence, the left child is processed first in both query trees. The variables X and Y are instantiated by the subquery of the left child of the query tree shown in Figure 6(b) and then they are used by the subquery of the right child.

*Query 2:*

```
select segment, X, Y
from video
where ((west(X,Y) before disjoint(X,Y)) and
       ((appear(Y) before touch(X,Y)) and
       (X != car1 and Z=project(X, [west(X,a)])))
```



(a)



(b)

Figure 7: (a) Initial query tree for *Query 2* and (b) Query tree for *Query 2* after internal node reordering

In the query tree, the children of the root 'AND' node are exchanged since the type of the left child is 'TEMPORAL' and the type of the right child is 'AND' in the initial query tree. The children of the 'AND' node, which is a child of the root node, are also exchanged since the type of the left child is 'TEMPORAL' and the type of the right child is 'AND' in the initial query tree (see Figure 7).

## 4.3 Leaf Node Reordering Algorithm

After the internal node reordering, the leaf nodes are reordered for each deepest internal node. Fact base statistics for each video is kept in a separate file. The number of each spatio-temporal relation in the video is stored in this file. Thus, the numbers of *south, northwest, southwest, equal, cover, inside, touch, disjoint, overlap, infrontof, behind, strictlyinfrontof, strictlybehind, touchfrombehind, touchedfrombehind* and *samelevel* facts are included in the file. These fact base statistics are used in the leaf node reordering algorithm. In this algorithm, the facts in the leaf nodes are sorted starting from the fact with the least number in fact base statistics file to the fact with the largest number. 'PPQ' and 'PRQ' type leaf nodes are reordered according to these statistics. These leaf nodes contain maximal subqueries that can be directly sent to the inference engine. Consequently, subquery trees for these maximal subqueries must be constructed to reorder leaf nodes. This construction is implemented within the query tree construction part. As a result, subquery trees for each maximal subquery in the 'PPQ' and 'PRQ' type leaf nodes are built and kept in a list data structure. The leaf node reordering algorithm is given in Figure 8.

This algorithm iterates on the query tree. Steps of the algorithm are as follows:

1. Find the 'PPQ' and 'PRQ' type leaf nodes.
2. Find the subquery trees of these nodes in the subquery list.
3. Reorder these subquery trees.
4. Get the content of the reordered subqueries.
5. Replace the contents of the leaf nodes with this content.

As it can be seen from the algorithm, only the condition parts of the 'PRQ' type leaf nodes are replaced. The functions used in the algorithm are explained in the sequel.

`FindPPQinList` function is used for locating the subquery tree of a particular leaf node in the subquery list (see Figure 9).

The `reorderAlg` function iterates on the subquery tree, which is located in the subquery tree list, and restructures this tree (see Figure 10). The algorithm first locates the highest 'AND'

```
LeafNodeReorder(QueryNode qnode,QueryTree qtree)
 // Iterate on the tree if node is not null
 if (qnode != NULL)
    begin
      type=qnode->Type()
      queryid=qnode->getQID(INORDER)
      // locate 'PPQ' and 'PRQ' leaf nodes
      if (type==PPQ or type==PRQ)
        begin
          // find the subquery tree of
          // the nodes in subquery list
          tmpppq=FindPPQinList(qtree, queryid)
          // reorder the subquery tree
          reorderAlg(tmpppq->ppqnode)
          // get the reordered subquery
          getSubquery(tmpppq->ppqnode)
          // set the content of the node
          if (type==PPQ)
             set content of qnode as subquery
          else if (type==PRQ)
             set content of the condition part of
             qnode as subquery
        end
     end
 // call the function recursively for left and right subtrees
 if (qnode->Left != NULL)
   LeafNodeReorder(qnode->Left,qtree)
 if (qnode->Right != NULL)
   LeafNodeReorder(qnode->Right,qtree)
```

Figure 8: Leaf node reordering algorithm

```
FindPPQinList(QueryTree qtree, int queryid)
   // locate the subquery tree of the leaf node with
   // id=queryid in the subquery list tmpppq
   tmpppq=qtree->headppq
   for (int i=1; i<qtree->ppqcount ; i++)
       if (queryid != tmpppq->queryid)
           tmpppq=tmpppq->nextppq
        else  break
```

Figure 9: The function that finds subquery tree of a leaf node

```
reorderAlg(QueryNode qnode)
  // Iterate on the subquery tree located
  // in the subquery tree list
  norecurse=0
  if (qnode!= NULL)
    begin
      type=qnode->Type
      // locate the highest 'AND' node on the subquery tree
      if (type==AND)
         if (qnode->Left != NULL and qnode->Right != NULL)
           begin
             ltype=qnode->Left->Type
             rtype=qnode->Right->Type
             // exchange left and right children
             // if the left child is 'NOT-OR' type
             // and the right child is 'AND' type
             if (ltype==NOT-OR and rtype==AND)
                exchange(qnode->Left, qnode->Right)
             // If children are 'PPQ' and 'AND' typed and
             // there is no 'NOT-OR' type node below these
             // children order the leaf nodes of this subtree
             // else if there is no 'NOT-OR' type node in the
             // right subtree put this subtree to left
             else if ( (ltype==AND and rtype==AND)
                       or(ltype==AND and rtype==PPQ)
                       or(ltype==PPQ and rtype==AND)
                       or(ltype==PPQ and rtype==PPQ) )

                     if (ThereIsNoOrNot(qnode)==1)
                        begin
                           OrderLeafNodes(qnode)
                           norecurse=1
                        end
                     else if (ThereIsNoOrNot(qnode->Right)==1)
                       exchange(qnode->Left, qnode->Right)
           end
      // call the function recursively for left and right
      // subtrees if a maximal 'AND' subtree is not located
      if (norecurse != 1)
       begin
          reorderAlg(qnode->Left)
          reorderAlg(qnode->Right)
       end
    end
```

Figure 10: The function that reorders the located subquery tree

```
ThereIsNoOrNot(QueryNode root)
 // return 0 if there is at least one ‘NOT-OR’
 // type node in the tree return 1 otherwise
 if (root->Left != NULL)
     begin
        if (root->Left->Type==NOT-OR)
            return 0
        if (ThereIsNoOrNot(root->Left)==0)
            return 0
     end
 if (root->Right != NULL)
     begin
        if (root->Right->Type==NOT-OR)
            return 0
        if (ThereIsNoOrNot(root->Right)==0)
            return 0
     end
 return 1
```

Figure 11: The function that finds if there is a ‘NOT-OR’ type node in a tree

type node in the subquery tree. If this node has left and right children and the left child is ‘NOT-OR’ type and the right one is ‘AND’ type, it exchanges the left and right nodes. If the children are ‘PPQ’ or ‘AND’ type and there is no ‘NOT-OR’ type node below these children, this subtree is called *maximal AND subtree* and it is reordered according to fact-base statistics. If the children are ‘PPQ’ or ‘AND’ type and there is at least one ‘NOT-OR’ type node below these children, the algorithm finds out whether the right child is a *maximal AND subtree* or not. If it is a *maximal AND subtree*, then it exchanges the right child with the left child. If the algorithm locates a *maximal AND subtree*, it does not recurse because it has already reordered all the nodes in the subtree. Otherwise, it recurses.

`ThereIsNoOrNot` function returns 0 if there is a ‘NOT-OR’ type node in a tree and returns 1 if all the nodes are ‘AND’ type (see Figure 11).

`OrderLeafNodes` function orders a *maximal AND subtree.* It first puts the leaf nodes into an array structure, then sorts the array according to the fact-base statistics and puts the leaf nodes back to the tree (see Figure 12).

`GetLeafNodes` function gets the leaf nodes of a tree and puts the contents and global variable counts of the nodes to an array structure to be used in the sorting procedure (see Figure 13).

```
OrderLeafNodes(QueryNode qnode)
 // get the leaf nodes of the maximal AND subtree
 // sort the leaf nodes according to the fact base statistics
 // put the leaf nodes back to the tree
 leafcounter=0
 GetLeafNodes(qnode,nodesarr)
 SortLeafNodes(nodesarr)
 leafcounter=0
 PutLeafNodes(qnode,nodesarr)
```

Figure 12: The function that orders leaf nodes

```
GetLeafNodes(QueryNode qnode,nodedata nodesarr[])
 // get the leaf nodes of the tree and put their contents
 // and global variable counts to the array nodesarr
 if (qnode->Left != NULL)
     if (qnode->Left->Type==PPQ)
         begin
           nodesarr[leafcounter].ncontent=qnode->Left->Content
           nodesarr[leafcounter].ppqflag=gvcount(qnode->Left)
           leafcounter++
         end
 if (qnode->Right != NULL)
     if (qnode->Right->Type==PPQ)
        begin
          nodesarr[leafcounter].ncontent=qnode->Right->Content
          nodesarr[leafcounter].ppqflag= gvcount(qnode->Right)
          leafcounter++
        end
 // call the function recursively for left and right subtrees
 if (qnode->Left != NULL)
   GetLeafNodes(qnode->Left, nodesarr)
 if (qnode->Right != NULL)
   GetLeafNodes(qnode->Right, nodesarr)
```

Figure 13: The function that gets leaf nodes

```
SortLeafNodes(nodedata nodesarr[])
 // sort the leaf nodes according to the fact base
 // statistics
 for (i=1; i<leafcounter; i++)
  begin
     for (j=i; j>0 and getnum(nodesarr[j])
                       <getnum(nodesarr[j-1]);j--)
          exchange(nodesarr[j],nodesarr[j-1])
     // put the relations that query an inequality
     // between any two objects in the video
     // to the end of the order
     for (i=0; i<leafcounter; i++)
        if ((nodesarr[i].ncontent.find("!=")) and
            (nodesarr[i].ppqflag>1))
          begin
            shift nodesarr left starting from i+1 to j
            put nodesarr[i] to the end of the array nodesarr
          end
  end
```

Figure 14: The function that sorts the leaf nodes

`SortLeafNodes` function sorts the leaf nodes according to the fact-base statistics. It orders the relations in the increasing number of statistics (see Figure 14). The `getnum` function gets the statistics of a particular relation from the statistics file of the video. After sorting the relations according to the statistics, the function puts the relations that query an inequality between any two objects in the video to the end of the order.

`PutLeafNodes` function puts the elements of an array structure to the leaf nodes of a tree. Hence, the nodes of the unsorted tree are replaced with the sorted nodes (see Figure 15).

### 4.3.1 Examples

Some query examples for leaf node reordering algorithm are given in this part. The initial queries and the queries after leaf node reordering according to the fact base statistics are shown. The relations in the query examples are reordered according to (*south facts* < *samelevel facts* < *west facts* < *overlap facts* < *disjoint facts* < *appear facts*).

*Query 1:*

```
select segment, X, Y
from video
```

```
PutLeafNodes(QueryNode qnode,nodedata nodesarr[])
 // put the elements of the array nodesarr to the
 // leaf nodes of the tree with the root qnode
 if (qnode->Left != NULL)
   begin
     if (qnode->Left->Type==PPQ)
       begin
        qnode->Left->setContent(nodesarr[leafcounter].ncontent)
        leafcounter++
       end
     PutLeafNodes(qnode->Left,nodesarr)
   end
 if (qnode->Right != NULL)
   begin
     if (qnode->Right->Type==PPQ)
       begin
        qnode->Right->setContent(nodesarr[leafcounter].ncontent)
        leafcounter++
       end
     PutLeafNodes(qnode->Right,nodesarr)
   end
```

Figure 15: The function that puts the elements to the leaf nodes

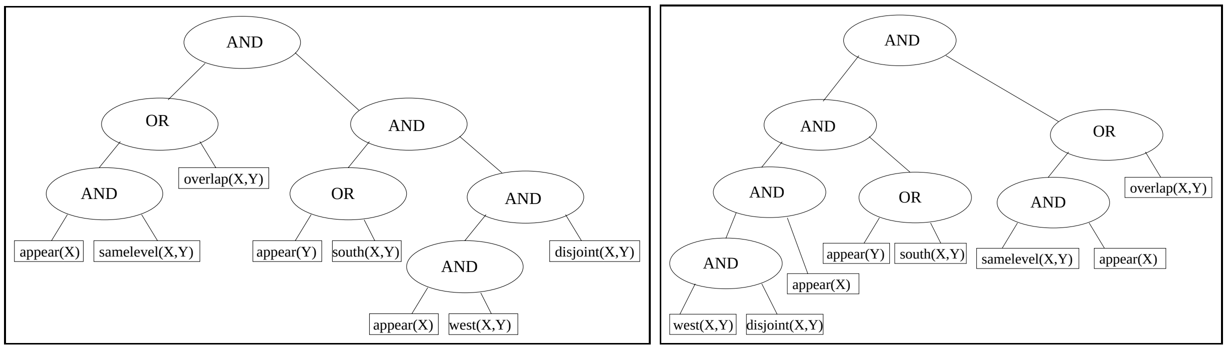
```
where  ((appear(X) and samelevel(X,Y)) or overlap(X,Y))
       and ((appear(Y) or south(X,Y)) and
          (appear(X) and west(X, Y) and disjoint(X,Y)))
```

*Query 1 after leaf node reordering:*

```
select segment, X, Y
from video
where ((west(X, Y) and disjoint(X,Y) and appear(X))
        and (appear(Y) or south(X,Y)))
      and ((samelevel(X,Y) and appear(X)) or overlap(X,Y))
```

The initial subquery tree for *Query 1* and the subquery tree for *Query 1* after leaf node reordering, which are located in the subquery tree list, are shown in Figure 16.

The leaf node reordering algorithm exchanges the left and right children of the root node first, then it exchanges the left and right children of the new left child which is an 'AND' node. After these steps, the algorithm reorders the relations in the left child of this 'AND' node according to



(a) (b)

Figure 16: (a) Initial subquery tree for *Query 1* and (b) Subquery tree for *Query 1* after leaf node reordering

the ordering rule *west facts* < *disjoint facts* < *appear facts.* The algorithm processes the right child of the root which is now an 'OR' node after the above steps. It reorders the relations in the left child of this 'OR' node according to the rule *samelevel facts* < *appear facts.*

*Query 2:*

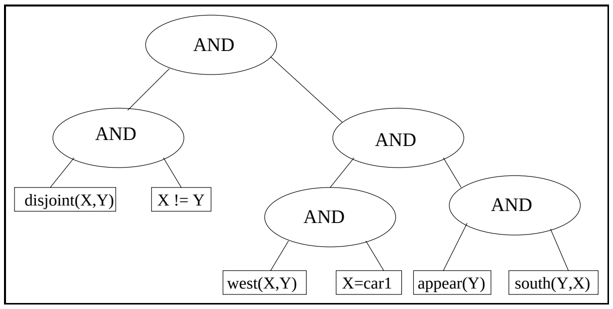
```
select segment, X, Y
from video
where disjoint(X,Y) and X != Y and west(X,Y)
       and X=car1 and appear(Y) and south(Y,X)
```

*Query 2 after leaf node reordering:*

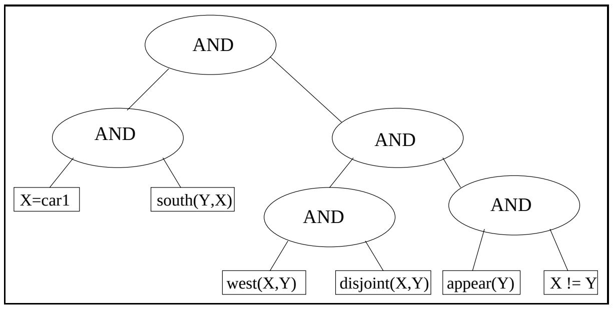
```
select segment, X, Y
from video
where  X=car1 and south(Y,X) and west(X,Y)
       and disjoint(X,Y) and appear(Y) and X != Y
```

The initial subquery tree for *Query 2* and the subquery tree for *Query 2* after leaf node reordering, which are located in the subquery tree list, are shown in Figure 17.

The relations in Query 2 are reordered as it can be seen from the second query according to the ordering *south facts* < *west facts* < *disjoint facts* < *appear facts.* The equality relations are executed at the beginning of the condition part and the inequality relations between variable objects are executed at the end.



(a)



(b)

Figure 17: (a) Initial subquery tree for *Query 1* and (b) Subquery tree for *Query 1* after leaf node reordering

## 5 Performance Results

In this section, the performance results obtained for the proposed query optimization algorithms are presented. Our performance tests have been conducted on an example video that was extracted from television news. The sample video contains 16,351 frames and 98 salient objects. The tests have been carried out on Linux environment using the query processor of BilVideo implemented in C++.

### 5.1 Fact-Base Statistics

The fact-base of the example video is created using the fact-extractor tool of BilVideo (Dönderler et al., 2002a). In this process, first of all, the objects in the sample video are defined, then the spatio-temporal relationships between these objects in each frame are calculated by our fact-extractor tool and stored in a fact base. Some example facts from our fact base are shown in Figure 18.

The statistics of the video are given in Table 1. This statistical information contains the number of facts in the fact base for each type of relation. These statistics are used in the optimization algorithm to reorder the leaf nodes.

### 5.2 Performance Results

Five query sets were used in the performance tests. The first query set was used for testing the *Leaf Node Reordering* algorithm. The second set was used for testing the *Internal Node*

```
// Directional Relations
west(tank1,car1,259).
west(car1,car2,259).
south(palestinianofficer4,officialcar,579).

//Topological Relations
disjoint(car2,car5,303).
disjoint(car1,car5,303).
overlap(officialcar,powell,503).
overlap(bodyguard1,officialcar,503).

// 3-D Relations
touchfrombehind(erbil,prizecheck,14434).
strictlyinfrontof(powell,officialcar,535).
strictlyinfrontof(powell,officialcar,542).
infrontof(policevehicle2,tank10,3076).
infrontof(vuralsavas,ozbek,10491).

// Appear Facts
appear(tank1,[[259,286]]).
appear(car1,[[259,365]]).
```

Figure 18: Example facts from our fact base

*Reordering* and *Leaf Node Reordering* algorithms together. The third and fourth sets were constructed for testing the algorithms on different reorderings of the same query. Finally, the fifth set was used for testing the same query on different sizes of fact bases and result sets. The query sets can be found in Appendix A. *Optimization overhead* given in the results specifies the time that the optimization process takes and *performance gain* is formulated as follows:

$$performance\ gain = \frac{(processing\ time\ without\ opt. - processing\ time\ with\ opt.)}{processing\ time\ without\ opt.}. \quad (1)$$

The first set of results are given in Table 2. These results show that leaf node reordering algorithm enhances the performance of the query processor. There are different amounts of performance gain for each query in the set. This is because the performance gain depends on the size of the query (i.e. the number of nodes in the query tree of the query), and the degree of difference between the initial query tree and the optimal query tree. The sizes of the first, ninth, eleventh and twelfth queries are small; therefore, their performance gains are at most 0.21. If the size of the query is small, the performance gain is also small compared to the larger queries.

Leaf node reordering algorithm reduces the processing cost because the relations in the leaf

Table 1: The statistics of the fact base

| *Type of relation* | *Number* |
|---|---:|
| west | 1055 |
| east | 1055 |
| south | 206 |
| northwest | 0 |
| southwest | 0 |
| disjoint | 1682 |
| overlap | 1235 |
| inside | 0 |
| appear | 10234 |
| touch | 9 |
| touchfrombehind | 37 |
| strictlyinfrontof | 184 |
| infrontof | 276 |
| samelevel | 487 |

nodes are ordered starting from the relation with the smallest size of output to the relations with larger sized outputs. Thus, the unbound variables in the nodes are first bound with smaller sets of values and relations with constant parameters are executed earlier. This results in an increase in performance. The second set of results are given in Table 3.

These results show that the overall query optimization algorithm improves the query processing performance. The factors that affect the results obtained with the leaf node reordering algorithm discussed above also affect those with the whole optimization process.

The query optimization algorithms reduce the processing cost because the subqueries with larger selectivities are processed before the subqueries with smaller selectivities. For example, if the children of an 'and' node are 'or' and 'and' type internal nodes, the 'and' type child is processed before the other, which results in a considerable gain in performance.

The performance gain depends on the size and complexity of the query. Another factor affecting the performance is the degree of difference between the initial query tree and the optimal query tree. The third and fourth performance tests were conducted using different reorderings of the same query. The query tree converges to the optimal query tree starting from the first query. The third result set that uses a simple Prolog query is given in Table 4. The fourth result set that uses a larger query tree is given in Table 5.

These two result sets show that when the query tree converges to the optimal query tree, the

Table 2: Leaf node reorder algorithm test results (msecs)

| *query* | *time without opt.* | *time with opt.* | *optimization overhead* | *performance gain* |
|---|---|---|---|---|
| 1 | 310 | 263 | 1.0 | 0.15 |
| 2 | 1002 | 609 | 1.0 | 0.39 |
| 3 | 512 | 264 | 1.0 | 0.48 |
| 4 | 490 | 291 | 1.0 | 0.41 |
| 5 | 508 | 217 | 1.0 | 0.57 |
| 6 | 423 | 261 | 1.0 | 0.38 |
| 7 | 2027 | 259 | 1.0 | 0.87 |
| 8 | 752 | 708 | 2.0 | 0.06 |
| 9 | 303 | 258 | 1.0 | 0.15 |
| 10 | 2030 | 1603 | 3.0 | 0.21 |
| 11 | 225 | 214 | 1.0 | 0.05 |
| 12 | 270 | 215 | 1.0 | 0.20 |

performance gain of the optimization algorithm decreases. This also justifies the correctness of the optimization algorithm.

The last performance test was conducted for investigating the impact of the query result set size on the performance gain. A query was selected and its result set size was decreased by decreasing the fact base size at each step. The results of this test are presented in Table 6. As it can be seen from the performance results, when the size of query result set decreases, the performance gain of the query does not change much, and it is within the range of 0.64-0.71.

The performance test results prove that the query optimization method implemented for BilVideo improves the performance of the query processor. Since the performance gain is observed to decrease when the query tree converges to the optimal query tree, it can be said that the reordering heuristics used by the algorithm are correct. As a conclusion, it is shown that processing more selective subqueries contained in the internal nodes and leaf nodes of the query tree earlier than the others is very useful in optimizing query processing times for spatio-temporal queries in video database systems.

## 5.3 Examples

Some queries selected from the set of queries used in the performance tests are discussed in this part. The initial query trees and the query trees after optimization are shown for each query.

Table 3: Query optimization algorithm test results (msecs)

| *query* | *time without opt.* | *time with opt.* | *optimization overhead* | *performance gain* |
|---|---|---|---|---|
| 1 | 690 | 212 | 1.0 | 0.69 |
| 2 | 958 | 530 | 2.0 | 0.45 |
| 3 | 532 | 270 | 1.0 | 0.49 |
| 4 | 327 | 267 | 1.0 | 0.18 |
| 5 | 644 | 283 | 2.0 | 0.56 |
| 6 | 639 | 344 | 1.0 | 0.46 |
| 7 | 545 | 337 | 1.0 | 0.38 |
| 8 | 274 | 214 | 1.0 | 0.22 |
| 9 | 261 | 211 | 1.0 | 0.19 |
| 10 | 985 | 286 | 1.0 | 0.71 |
| 11 | 302 | 213 | 2.0 | 0.29 |
| 12 | 845 | 283 | 2.0 | 0.67 |

Table 4: Convergence to the optimal query tree; first test results (msecs)

| *query* | *time without opt.* | *time with opt.* | *optimization overhead* | *performance gain* |
|---|---|---|---|---|
| 1 | 1327 | 256 | 2.0 | 0.81 |
| 2 | 341 | 256 | 2.0 | 0.25 |
| 3 | 305 | 255 | 1.0 | 0.16 |
| 4 | 253 | 253 | 1.0 | 0.00 |

*Query 1:*

```
select segment, X, Y
from video
where (west(X,Y) and disjoint(X,Y) and X != car1
       or Z = project(X,[west(X, car1)])) and (west(X,Y)
       and T =  project(X,[west(X, car1)]))
```

The initial query tree of Query 1 (Figure 19 (a)) is processed in 985 milliseconds and the optimized query tree (Figure 19 (b)) is processed in 286 milliseconds. Consequently, the performance gain is 71%.

Table 5: Convergence to the optimal query tree; second test results (msecs)

| *query* | *time without opt.* | *time with opt.* | *optimization overhead* | *performance gain* |
|---|---|---|---|---|
| 1 | 1306 | 218 | 2.0 | 0.83 |
| 2 | 1213 | 220 | 1.0 | 0.82 |
| 3 | 663 | 218 | 2.0 | 0.67 |
| 4 | 647 | 219 | 3.0 | 0.66 |
| 5 | 563 | 220 | 2.0 | 0.61 |
| 6 | 345 | 222 | 2.0 | 0.36 |
| 7 | 324 | 219 | 2.0 | 0.32 |
| 8 | 219 | 219 | 2.0 | 0.00 |

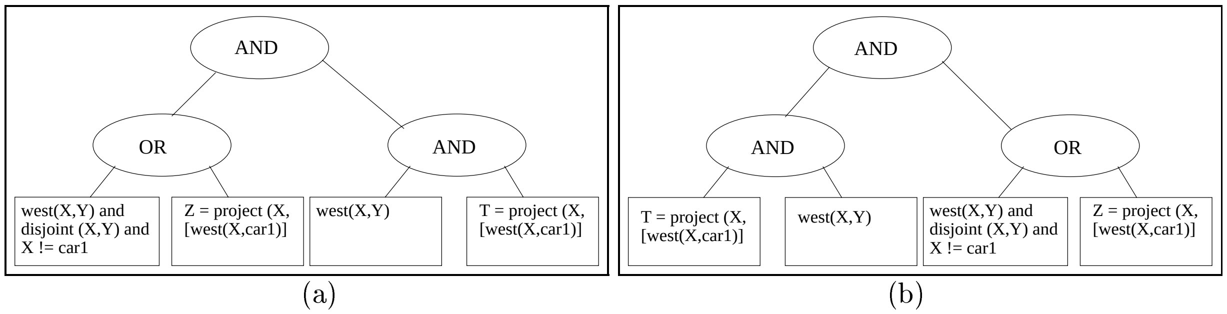

AND
OR
AND
west(X,Y) and disjoint (X,Y) and X != car1
Z = project (X, [west(X,car1)]
west(X,Y)
T = project (X, [west(X,car1)]
(a)
AND
AND
OR
T = project (X, [west(X,car1)]
west(X,Y)
west(X,Y) and disjoint (X,Y) and X != car1
Z = project (X, [west(X,car1)]
(b)

Figure 19: (a) Initial query tree for *Query 1* and (b) Query tree for *Query 1* after optimization

*Query 2:*
```
select segment, X, Y
from video
where (samelevel(X,Y) before disjoint(X,Y)) and
       (infrontof(X,Y) and X != car1 and tr(X, [[west], [1]]))
```

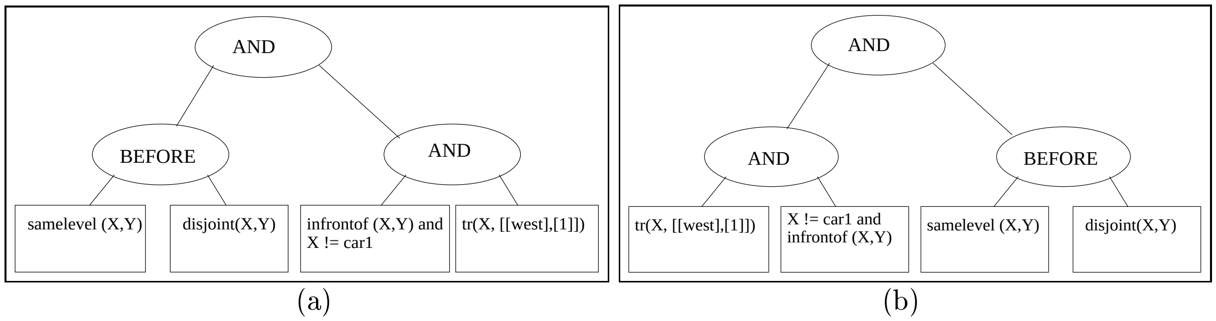


Figure 20: (a) Initial query tree for *Query 2* and (b) Query tree for *Query 2* after optimization

The initial query tree of Query 2 (Figure 20 (a)) is processed in 845 milliseconds and the optimized query tree (Figure 20 (b)) is processed in 283 milliseconds. Thus, the performance gain is 67%.

Table 6: Query result set size parameter test results (msecs)

| *size of result set* | *time without opt.* | *time with opt.* | *performance gain* |
|---|---|---|---|
| 133 | 2533 | 786 | 0.69 |
| 120 | 2259 | 713 | 0.68 |
| 105 | 2067 | 665 | 0.68 |
| 94 | 2013 | 632 | 0.69 |
| 85 | 1960 | 616 | 0.69 |
| 74 | 1673 | 538 | 0.68 |
| 65 | 1399 | 449 | 0.68 |
| 45 | 1275 | 379 | 0.70 |
| 34 | 1209 | 353 | 0.71 |
| 27 | 830 | 281 | 0.66 |
| 20 | 688 | 251 | 0.64 |
| 11 | 669 | 231 | 0.65 |
| 2 | 650 | 208 | 0.68 |

## 6 Conclusions and Future Work

Query processing is essential for retrieving data from database management systems and has been explored in the last 30 years in the contest of relational and object-oriented database management systems. Query optimization constitutes an important part of query processing, and it is a promising research area since the amount of data that can be managed by database systems is growing rapidly and new data types are becoming widely used. Besides, new types of database management systems such as multimedia databases require new techniques for query processing and query optimization.

In this paper, we have presented a query optimization strategy for video database systems, which was implemented on a particular system, BilVideo. The proposed optimization method has two basic parts: *internal node reordering* and *leaf node reordering*. The children of the internal nodes of the query tree of a given query are reordered using the internal node reordering algorithm which places more selective children to the left of their parents. The contents of the Prolog and Project type leaf nodes are reordered using the leaf node reordering algorithm, which makes use of statistical information to sort the relations forming the contents of the leaf nodes. Therefore, our optimization method reorders the query tree along two dimensions that results in a considerable improvement in performance. The performance tests conducted on the query processor justify the efficiency and correctness of the query optimization algorithms, internal node reordering and leaf node reordering.

Currently, the proposed optimization algorithms are used by a query processor that uses linear processing methods. The algorithms can be adapted to a parallel query processor as a future work, which can result in an even better performance. Another future work can be the use of genetic algorithms in query optimization of BilVideo as they are becoming widely used and accepted method for new and difficult optimization problems. This method must propose a fitness value function for the query trees in the solution space and adapt cross-over and mutation operations to produce efficient query trees.

## References

Atnafu, S., Brunie, L., Kosch, H., 2001. Similarity-based operators and query optimization for multimedia database systems, In: Proc. of Int. Database Eng. and App. Symp., 346-355.

Chang, N.S., Fu, K.S., 1980. Query by pictorial example, IEEE Trans. on Software Eng., SE6, 519-524.

Chang, S., Chen, W., Meng, H.J., Sundaram, H., Zhong, D., 1997. VideoQ: an automated content-based video search system using visual cues, In: Proc. of ACM Multimedia Conference, 313-324, Seattle, WA.

Chaudhuri, S., 1998. An overview of query optimization in relational systems, In: Proc. of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 34-43.

Chu, W.W., Cardenas, A.F., Taira, R.K., 1995. A knowledge-based multimedia medical distributed database system - KMED, Information Systems, 20 (2), 75-96.

Dönderler, M.E., Ulusoy, Ö, Güdükbay, U., 2000. A rule-based approach to represent spatio-temporal relations in video data, In: Yakhno, T. (Ed.), Advances in Information Systems (ADVIS'00), Lecture Notes in Computer Science, 1909, 409-418, Springer, Berlin.

Dönderler, M.E., Ulusoy, Ö, Güdükbay, U., 2002a. A rule-based video database system architecture, Information Sciences, 143 (1-4), 13-45.

Dönderler, M.E., Ulusoy, Ö, Güdükbay, U., 2002b. Rule-based spatio-temporal query processing for video databases, Technical Report, BU-CE-0210, Bilkent University, Dept. of Computer Eng. (also submitted to a journal).

Flickner, M., Sawhney, H., Niblack, W., Ashley, J., Huang, Q., Dom, B., Gorkani, M., Hafner, J., Lee, D., Petkovic, D., Steele, D., Yanker, P., 1995. Query by image and video content: the QBIC system, IEEE Computer, 28, 23-32.

Garofalakis, M.N., 1998. Query scheduling and optimization in parallel and multimedia databases, Ph.D. Thesis, University of Wisconsin, Madison, Wisconsin.

Guting, R.H., Bohlen, M.H., Ervig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M., Vazirgiannis, M., 2000. A foundation for representing and querying moving objects, ACM Trans. on Database Sys., 25 (1), 1-42.

Jarke, M., Koch, J., 1984. Query optimization in database Systems, ACM Computing Surveys, 16 (2), 111-152.

Li, J.Z., Özsu, M.T., Szafron, D., 1997. Modeling of moving objects in a video database, In: Proc. of IEEE Multimedia Computing and Systems, 336-343, Ottawa, Canada.

Mahalingam, L.P., Candan, K.S., 2001. Query optimization in the presence of top-k predicates, In: Proc. of Workshop on Multimedia Information Systems (MIS'01), 31-40, Capri, Italy.

Nabil, M., Ngu, A.H., Shepherd, J., 2001. Modeling and retrieval of moving objects, Multimedia Tools and Apps., 13, 35-71.

Oomoto E., Tanaka, K., 1993. OVID: design and implementation of a video object database system, IEEE Trans. on Knowl. and Data Eng., 5, 629-643.

Sistla, A.P., Wolfson, O., Chamberlain, S., Dao, S., 1997. Modeling and querying moving objects, In: Proc. of IEEE Data Engineering Conf., 422-432.

Soffer, A., Samet, H., 1999. Query processsing and optimization for pictorial query trees, In: Huijsmans D.P., Smeulders, A.W.M. (Eds.) Visual Information and Information Systems (VISUAL'99), Lecture Notes in Computer Science, 1614, 60-67, Springer, Berlin.

# A Appendix: Query Sets Used in Performance Experiments

## A.1. Query Set Used to Test Leaf Node Reorder Algorithm:

1.
   ```
   select segment, X, Y
   from 1
   where disjoint(X,Y) and south(X,Y)
   ```

2.
   ```
   select segment, X, Y
   from 1
   where appear(X) and west(X,Y)
   and disjoint(X,Y)
   ```

3.
   ```
   select segment, X, Y
   from 1
   where disjoint(X,Y) and west(X,Y)
   and X=car1
   ```

4.
   ```
   select segment, X, Y
   from 1
   where west(X,Y) and disjoint(X,Y)
   and south(X,Y)
   ```

5.
   ```
   select segment, X, Y
   from 1
   where disjoint(X,Y) and X != Y and
   west(X,Y) and X=car1 and appear(Y)
   and south(Y,X)
   ```

6.
   ```
   select segment, X, Y
   from 1
   where disjoint(X,Y) and west(tank1,car1)
   and X=car1 and appear(Y) and south(Y,X)
   ```

7.
   ```
   select segment, X, Y
   from 1
   where appear(Y) and west(X,Y) and south(Y,X)
   and X=tank1 and west(tank1,car1)
   ```

8.
   ```
   select segment, X, Y
   from 1
   where west(X,Y) and appear(X) and overlap(X,Y)
   ```

9.
   ```
   select segment, X, Y
   from 1
   where west(A,B) and touch(X,Y)
   ```

10.
    ```
    select segment, X, Y
    from 1
    where (samelevel(X,Y) and appear(X) and
    overlap(X,Y)) or (appear(X) and
    west(X, Y) and disjoint(X,Y))
    ```

```
11. select segment
    from all
    where Z = project(X, [disjoint(X, car1) and
    west(X,tank1) and south(car1,tank1)])

12. select segment
    from all
    where Z = project(X, [west(X, car1) and
    disjoint(X,tank1) and south(X,car2)])
```

**A.2. Query Set Used to Test Query Optimization Algorithm:**

```
 1. select segment, X, Y
    from 1
    where (west(X,Y) and disjoint(X,Y) and
    X != Y or Z = project(X,[west(X, a)])) and
    (west(X,Y) and X=car1 and appear(Y) and south(Y,X))

 2. select segment, X, Y
    from 1
    where (west(X,Y) and disjoint(X,Y) and
    X != car1 or Z = project(X,[west(X, car1)]))
    and (west(X,Y) before south(Y,X))

 3. select segment, X, Y
    from 1
    where (west(X,Y) before disjoint(Y,X))
    and (X != car1 and Z = project(X,[west(X, car1)]))

 4. select segment
    from all
    where tr(X, [[west], [1]]) and
    Y = project(X, [west(X, car1)])

 5. select segment, X, Y
    from 1
    where west(X,Y) and disjoint(X, Y) and
    X != car1 and Z = project(X,[west(X, car1)])

 6. select segment, X, Y
    from 1
    where west(X,Y) and disjoint(X, Y)
    and X != car1 and tr(X, [[west],[1]])

 7. select segment, X, Y
    from 1
    where west(X,Y) and tr(X, [[west],[1]])

 8. select segment
    from all
    where Y = project(X, [west(X, car1)]) and
    Z = project(X, [south(X,car1) and west(X,tank1)
    and disjoint(X, car1)])
 9. select segment
    from all
    where tr(X, [[west], [1]]) and
    tr(car3, [[west,north], [10,10]])

10. select segment, X, Y
    from 1
    where (west(X,Y) and disjoint(X,Y) and X != car1 or
    Z = project(X,[west(X, car1)])) and (west(X,Y) and
    T = project(X,[west(X, car1)]))

11. select segment, X, Y
    from 1
    where (west(X,Y) and touch(X, Y) and X != car1 or
    Z = project(X,[west(X, tank1)])) and (disjoint(X,Y)
    and overlap(X,Y) and Y != car2)

12. select segment, X, Y
    from 1
    where (samelevel(X,Y) before disjoint(X,Y)) and
    (infrontof(X,Y) and X != car1 and tr(X, [[west], [1]]))
```

**A.3. First Query Set that Tests the Convergence of the Initial Query Tree to the Optimal Query Tree:**

```
1. select segment, X, Y
   from 1
   where appear(X) and disjoint(X,Y) and south(X,Y)

2. select segment, X, Y
   from 1
   where disjoint(X,Y) and appear(X) and south(X,Y)

3. select segment, X, Y
   from 1
   where disjoint(X,Y) and south(X,Y) and appear(X)

4. select segment, X, Y
   from 1
   where south(X,Y) and disjoint(X,Y) and appear(X)
```

**A.4. Second Query Set that Tests the Convergence of the Initial Query Tree to the Optimal Query Tree:**

```
1. select segment, X, Y
   from 1
   where (disjoint(X,Y) and west(X,Y) and X != Y or
   Z = project(X,[west(X,a)])) and (appear(Y) and
   west(X,Y) and south(Y,X) and X=car1)

2. select segment, X, Y
   from 1
   where (west(X,Y) and disjoint(X,Y) and X != Y or
   Z = project(X,[west(X,a)])) and (appear(Y) and
   west(X,Y) and south(Y,X) and X=car1)

3. select segment, X, Y
   from 1
   where (west(X,Y) and disjoint(X,Y) and X != Y or
   Z = project(X,[west(X,a)])) and (west(X,Y) and
   appear(Y) and south(Y,X) and X=car1)

4. select segment, X, Y
   from 1
   where (west(X,Y) and disjoint(X,Y) and X != Y or
   Z = project(X,[west(X,a)])) and (west(X,Y) and
   south(Y,X) and appear(Y) and X=car1)

5. select segment, X, Y
   from 1
   where (west(X,Y) and disjoint(X,Y) and X != Y or
   Z = project(X,[west(X,a)])) and (X=car1 and
   south(Y,X) and west(X,Y) and appear(Y))

6. select segment, X, Y
   from 1
   where (west(X,Y) and appear(Y) and south(Y,X) and
   X=car1) and (west(X,Y) and disjoint(X,Y) and
   X != Y or Z = project(X,[west(X,a)]))

7. select segment, X, Y
   from 1
   where (west(X,Y) and south(Y,X) and X=car1 and
   appear(Y)) and (west(X,Y) and disjoint(X,Y) and
   X != Y or Z = project(X,[west(X,a)]))

8. select segment, X, Y
   from 1
   where (X=car1 and south(Y,X) and west(X,Y) and
   appear(Y)) and (west(X,Y) and disjoint(X,Y) and
   X != Y or Z = project(X,[west(X,a)]))
```