AN EFFICIENT QUERY OPTIMIZATION STRATEGY FOR SPATIO-TEMPORAL QUERIES IN VIDEO DATABASES

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Gülay Ünel

July, 2002

ABSTRACT

AN EFFICIENT QUERY OPTIMIZATION STRATEGY FOR SPATIO-TEMPORAL QUERIES IN VIDEO DATABASES

Gülay Ünel
M.S. in Computer Engineering
Supervisors: Assoc. Prof. Dr. Özgür Ulusoy and
Assist. Prof. Dr. Uğur Güdükbay
July, 2002

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 is 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 thesis, query optimization problem in video database systems is outlined and a query optimization strategy is proposed as a solution to this problem. Reordering algorithms, to be applied on query execution tree, are also described. Finally, the performance results obtained by testing the proposed algorithms are presented.

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

ÖZET

VİDEO VERİTABANLARINDA YERLEŞİM-ZAMAN SORGULARI İÇİN ETKİLİ BİR SORGU OPTİMİZASYON STRATEJİSİ

Gülay Ünel
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticileri: Doç. Dr. Özgür Ulusoy and
Yrd. Doç. Dr. Uğur Güdükbay
Temmuz, 2002

Mültimedya veritabanı yönetim sistemlerine olan ilgi büyük hacimlerde mültimedya verilerini saklama ihtiyacından dolayı hızla artmıştır. Sorgu işlemcisi, bir mültimedya veritabanı sisteminin önemli yapı taşlarından biridir ve sorguları verimli bir şekilde yanıtlayabilmek için sorgu işlemcisine yerleştirilmiş bir sorgu eniyileyicisine ihtiyaç vardır. Sorgu optimizasyonu problemi konvansiyonel veritabanları için kapsamlı olarak araştırılmış olup, mültimedya veritabanı sistemleri için yeni bir araştırma alanıdır. Sorgu işleme stratejilerindeki farklılıklardan dolayı mültimedya veritabanı sistemlerinde kullanılan sorgu optimizasyon teknikleri, geleneksel veritabanlarında kullanılanlardan farklıdır. Bu tezde, video veritabanı sistemlerindeki sorgu optimizasyon problemi ana hatlarıyla ele alınmış ve bu probleme çözüm olarak bir sorgu optimizasyon stratejisi önerilmiştir. Ayrıca, sorgu çalışma ağacına uygulanacak sıralama algoritmaları tanımlanmıştır. Son olarak, önerilen algoritmaların test edilmesi sonucu elde edilmiş olan performans sonuçları sunulmuştur.

Anahtar sözcükler: video veritabanları, sorgu optimizasyonu, sorgu ağacı, video verilerini sorgulama.

Acknowledgement

I would like to express my special thanks and gratitude to my supervisors Assoc. Prof. Dr. Özgür Ulusoy and Assist. Prof. Dr. Uğur Güdükbay for their concern in the supervision of the thesis.

I would like to express my gratitude to Assist. Prof. Dr. Attila Gürsoy and Assist. Prof. Dr. İbrahim Körpeoğlu for their interest to the subject matter and spending their time for reading and reviewing the thesis.

I would like to acknowledge the support of Turkish Scientific and Technical Research Council (TÜBİTAK).

I would like to express my special thanks to Mehmet Emin Dönderler for his support and patience in all stages of the thesis research.

I thank to my spouse Cüneyt, my brother Semih, my mother and father for their support.

Finally, I would like to express my special thanks and gratitude to my manager Dr. Seyit Koçberber for all his support in my master's study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The interest for multimedia database systems has grown rapidly with the advances in computer technology. The research on content-based image retrieval by visual features (color, shape and texture) and keywords [4, 5] has progressed in time towards video databases dealing with spatio-temporal and semantic features of video data. First, the techniques devised for image retrieval were used for supporting content-based video retrieval. These techniques assumed the video as a consecutive sequence of images ordered in time. Some video database systems such as VideoQ, KMED, QBIC and OVID [6, 7, 5, 8] were implemented. Querying video objects by motion properties has also been studied [16, 17, 18, 19].

Building blocks for multimedia database systems are multimedia data model, multimedia storage management, query interface, and query processing and retrieval. Data models used in multimedia Database Management Systems (DBMSs) are different from those used in conventional DBMSs, so 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, like image, audio and video. Extensions to the conventional query languages are required that take into account of the particular characteristics of multimedia data. In addition to these, different query optimization techniques are required to be implemented and integrated to the system.

Success of a database system depends on the effectiveness of the query optimization module 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 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 any optimization algorithm is to find a member of S that has the minimum cost.

Query optimization has been a challenging research area starting from the beginning of the relational database management systems. A summary of the research efforts on query optimization and other related concepts in database systems can be found in [10].

In this thesis, we study the query optimization problem in multimedia database systems. Our work concentrates on reordering of query trees in processing queries in a multimedia database system to achieve the minimum cost. We propose algorithms used for reordering query trees. The goal of the optimization algorithms is to change the order of processing 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. The query optimization module contains two types of reorderings for query trees to ensure more efficient processing of queries. The first type is *internal node reordering*, which reconstructs the query tree by reordering the children of internal nodes. The second type is *leaf node reordering*, which restructures the query contents of the leaf nodes of the query tree. The query optimization algorithms are implemented as a part of the query processor of a video database system and tested using sample videos.

The work done in this thesis constitutes a part of a video database system, BilVideo, developed by Dönderler et al. [1, 2, 3]. In this system, a rule-based spatio-temporal model for videos and a video query processor, which can answer spatial, temporal, trajectory, motion and object queries for videos, are proposed. The work done in this thesis is integrated into the query processor of BilVideo.

1.1 Organization of the Thesis

The remainder of the thesis is organized as follows. In Chapter 2, related work on multimedia query optimization is discussed. The video database system, into which query optimization module is integrated, is described in Chapter 3. In Chapter 4, our query optimization algorithms are presented. Performance results are discussed in Chapter 5. Conclusions of our work and future research directions are given in Chapter 6. Fact base of the example database and the query sets used in the experiments are given in Appendices A and B, respectively.

Chapter 2

Related Work

Basic principles of query optimization in database systems are explained by Jarke and Koch [10]. In their paper, a wide variety of approaches are proposed to improve the performance of query processing 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. These methods are presented in the framework of a general query evaluation procedure using relational calculus representation of queries. In addition to these methods, nonstandard query optimization issues are also discussed in the paper. According to Jarke and Koch, the goals of query transformation are: (1) the construction of a standardized starting point for query optimization (*standardization*), (2) the elimination of redundancy (*simplification*), and (3) the construction of expressions that are improved with respect to evaluation performance (*amelioration*). The transformation rules for the general query expressions referenced in the paper are also valid for our query expressions.

Chaudhuri [13] focuses primarily on the optimization of SQL queries in relational database systems. According to the paper, the two key components of the query evaluation component of an SQL database system are the query optimizer and the query execution engine. The paper discusses the System-R optimization framework, search space that is considered by optimizers, cost estimation and enumerating the search space. The basic cost estimation framework in System-R uses statistical summaries of data that have been stored. It also determines the statistical summary of the output data stream and estimated cost of executing the operation given an operator and the statistical summary for each of its input data streams. The idea of collecting statistical summaries for cost estimation is also used in our query optimization module.

The survey of query evaluation techniques for large databases by Graefe [11] describes query evaluation techniques for both relational and postrelational database systems, including iterative execution of complex query execution plans, the duality of sort- and hash- based set-matching algorithms, types of parallel query execution and their implementation, and special operators for emerging database application domains. According to the survey, query optimization is a special form of planning, and employing techniques from artificial intelligence such as plan representation, search including directed search and pruning, dynamic programming, branch-and-bound algorithms, etc.

Semantic query optimization for tree and chain queries by Sun and Yu [9] provides an effective and systematic approach for optimizing queries by appropriately choosing semantically equivalent transformations. Basically, there are two different types of transformations: transformations by eliminating unnecessary joins, and transformations by adding/eliminating redundant beneficial/nonbeneficial selection operations (restrictions). An algorithm is proposed by Sun and Yu to minimize the number of joins in tree queries. They claim that the important operations in semantic query optimization are the detection of a contradiction, the elimination of as many unnecessary joins as possible, and the addition/elimination of beneficial/nonbeneficial redundant restrictions.

Alternative plan generation methods for multiple query optimization by Menekse et al. [12] focus on generating a number of alternative plans in such a way that the sharing between queries is maximized and an optimal execution plan with minimal cost is obtained. They state that a global execution plan can be constructed by choosing one plan for each query and then merging these plans together. Two algorithms for alternative plan generation have been implemented, which are *pairwise transformation* and *complete transformation*. A new plan for alternative plan generation is also proposed to eliminate useless alternative plans by introducing a sharing factor concept.

The paper by Soffer and Samet [14] presents optimization methods for processing of pictorial queries specified by pictorial query trees. Their optimization strategy for computing the result of the pictorial query tree is to change 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 based on 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 that we used. 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 algorithms for other types of nodes such as internal nodes that contain operators are added.

Mahalingam and Candan propose techniques for performing query optimization in different types of databases, such as multimedia and Web databases, which rely on top-k predicates [15]. 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. So, 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 from the query execution order in our work.

Chapter 3

BilVideo: A Video DBMS

In this chapter, a video database system, BilVideo [1, 2, 3] to which the work in this thesis is integrated, is described. BilVideo is a video database management system that supports spatio-temporal and semantic 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. The organization of this chapter is as follows: The architecture of BilVideo is given in Section 3.1. The video query language of BilVideo is described in Section 3.2. The query types are presented in Section 3.3. Query processing issues in BilVideo are discussed in Section 3.4.

3.1 Video Database System Architecture

Figure 3.1 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.

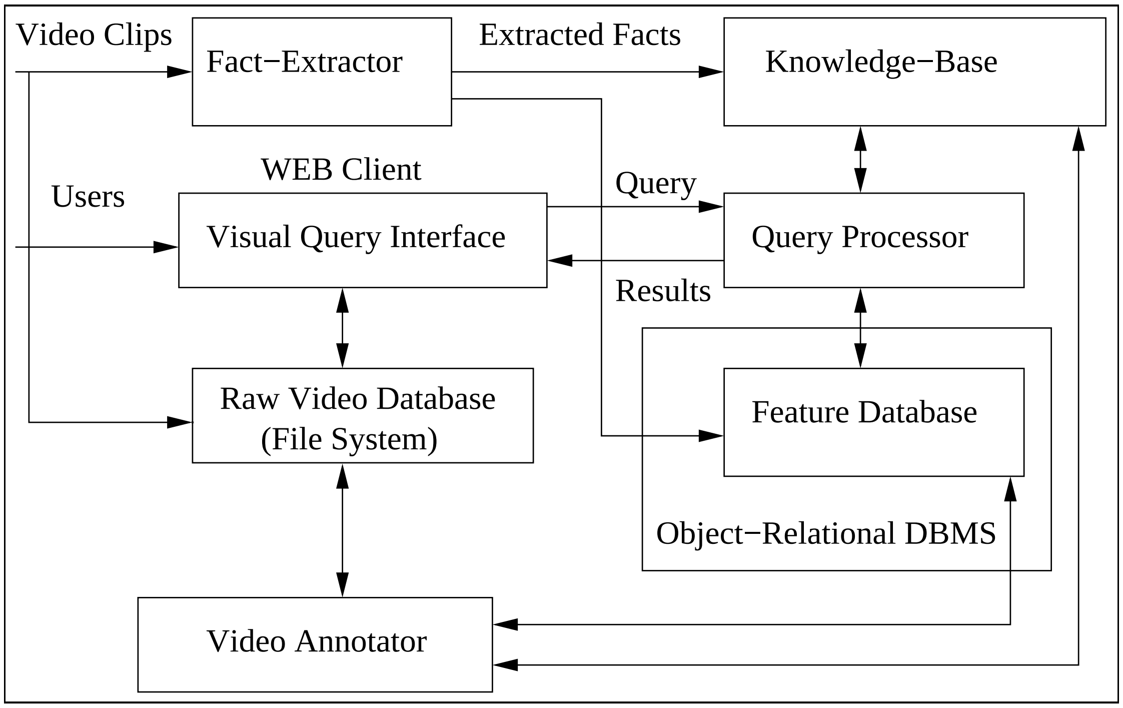

Figure 3.1: BilVideo database system architecture.

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[1] 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.

The rules used for querying the video data, called *query rules*, have associated frame numbers. A second set of rules, called *extraction rules*, was also created to work with frame intervals to extract spatio-temporal relations from video data. Extracted spatio-temporal relations are converted to facts with frame numbers of the keyframes in the knowledge-base. These facts are used by the query rules for query processing.

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

3.2 Video Query Language

The query language has four basic statements for retrieving information:

```
select video from all [where condition];
select video from videolist where condition;
select segment from range where condition;
select variable from range where 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 `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.

- *Supported operators:* The query language supports logical and temporal operators to be used in query conditions. Logical operators are *and, or* and *not.* Temporal operators are *before, meets, overlaps, starts, during, finishes* and their inverse operators. In addition to these, the query language has a trajectory-projection operator, *project*, which is used to extract subtrajectories of video objects on a given spatial condition. The language also has the operators '=' and '!=', which can be used for assignment and comparison.
- *Aggregate functions:* The aggregate functions of the query language are *average, sum* and *count.* They take a set of intervals (segments) as input and return a time value in minutes for each video clip satisfying given conditions.
- *External predicates:* The query language has a condition type *external*, which is defined for application-dependent predicates, called *external predicates.* External predicates are processed using the same methods with the

spatial predicates. If an external predicate is to be used, facts and/or rules related to the predicate should be added to the knowledge-base.

3.3 Query Types

The query language supports spatio-temporal, semantic and low-level queries. Different query types that can be specified by the query language are as follows:

- *Object queries:* This type of queries may be used to retrieve objects, along with video segments where the objects appear.
- *Spatial queries:* This type of queries may be used to query videos by spatial properties of objects defined with respect to each other. Supported spatial properties for objects can be grouped into mainly three categories: topological relations that describe neighborhood and incidence in 2D-space, directional relations that describe order in 2D- space, and 3D-relations that describe object positions on z-axis of the three dimensional space. There are eight distinct topological relations: *equal, cover, covered-by, inside, touch, disjoint, overlap* and, *contains.* Directional relations are *west, south, north, east, northwest, northeast, southwest* and, *southeast.* 3D relations are *infrontof, behind, strictlyinfrontof, strictlybehind, touchfrombehind, touchedfrombehind* and *samelevel.*
- *Similarity-based object-trajectory queries:* This type of queries may be used to query videos to find out the object and/or time interval of an object having a trajectory in the video to a given direction.
- *Temporal queries:* This type of queries is used to specify the order of occurrence for conditions in time.
- *Aggregate queries:* This type of queries may be used to retrieve statistical data about objects and events in video data. The three aggregate functions are *average*, *sum*, and *count.* These functions are useful in collecting statistical data for such applications as sports event analysis systems.

- *Low-level queries:* This type of queries is used to query video data by visual properties such as color, shape and texture.

- *Semantic queries:* This type of queries is used to query video data by semantic features. In the system, videos are partitioned into semantic units, which form a hierarchy. This semantic video hierarchy contains three levels: *video*, *sequence* and *scene*. *Videos* consist of *sequences*, and *sequences* consist of *scenes* that need not be consecutive in time. With this semantic data model, three types of queries will be answered which are *video*, *event/activity* and *object*.

3.4 Query Processing

Figure 3.2 illustrates how the query processor communicates with Web clients and the underlying system components to answer user queries. Figure 3.3 shows the phases of query processing for spatio-temporal 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 phases of query processing for spatio-temporal queries can be briefly described as follows.

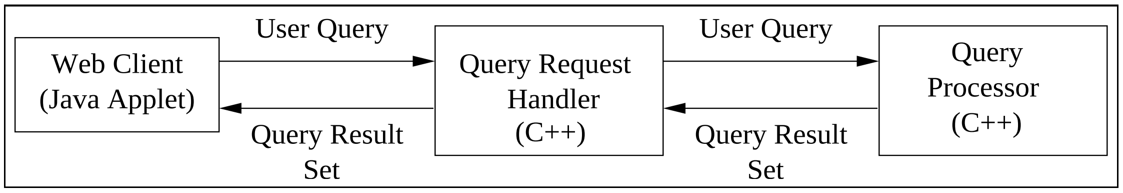

Figure 3.2: Web client - query processor interaction.

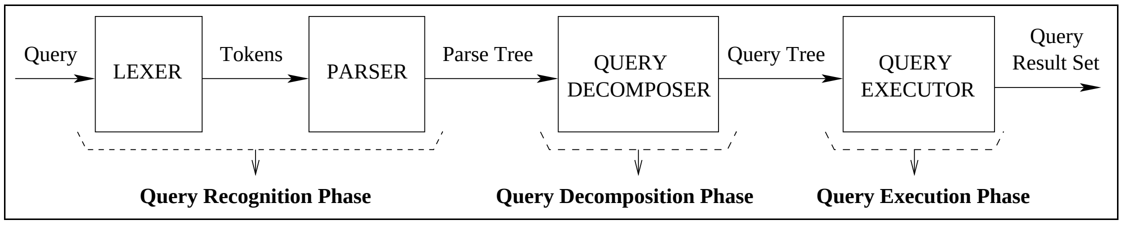

Figure 3.3: Query processing phases.

1. *Query recognition:* The lexical analyzer partitions a query into tokens, which are passed to the parser with possible values for further processing. The parser assigns structure to the resulting pieces and creates a parse tree to be used as a starting point for query processing. This phase is called *query recognition phase.*

2. *Query decomposition:* The parse tree generated after the query recognition phase is traversed in a second phase, which is called *query decomposition phase*, to construct a query tree. The query tree is constructed from the parse tree decomposing a query into three basic types of subqueries which are *Prolog subqueries* (directional, topological, 3D-relation, external predicate and object-appearance) that can be directly sent to the inference engine Prolog, *trajectory-projection subqueries* that are handled by the trajectory projector, and *similarity-based object-trajectory subqueries* that are processed by the trajectory processor. Maximal subqueries are subqueries that are formed by grouping prolog type predicates. A query is decomposed in such a way that minimum number of subqueries are formed.

3. *Query execution:* The input for the *query execution phase* is a query tree. This query tree is traversed in postorder in *query execution phase*, executing each subquery separately and performing interval processing so as to obtain the final set of results. Intermediate results returned by Prolog are further

processed in this phase and final answers to user queries are formed after interval processing.

Chapter 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 in which the more selective subqueries are executed earlier by the query processor. The query optimization process is outlined in Figure 4.1.

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. 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 right children of 'AND' nodes which are more selective than left children to the left of their parents. Leaf node reordering algorithm deals with the leaf

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

Figure 4.1: Query optimization process

nodes. Every leaf node in the query tree has a content which 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 which are connected by 'AND' operators according to their selectivity. More selective operations are executed earlier than the others by the reorderings of this algorithm.

This chapter is organized as follows: In Section 4.1, our query tree structure is explained. In Section 4.2, the internal node reordering algorithm is described. Finally, the leaf node reordering algorithm is presented in Section 4.3.

4.1 Structure of the Query Tree

In our multimedia database model, a query is represented by a query tree containing the spatio-temporal relationships between the data that is to be selected. 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 and/or time interval of an object having a trajectory in the video to a given direction. 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*, *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`.

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): Spatial subqueries processed by Prolog,
2. Trajectory Queries (TRQ): Object-trajectory subqueries, and
3. Project Queries (PRQ): Project subqueries.

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 left children 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 4.2.

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

- The '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, they are found out to be the least selective compared to the other nodes.
- The 'AND' type child nodes must be on left if the other child is 'TEMPORAL' type. This is because of the fact that 'AND' type nodes are processed faster than the 'TEMPORAL' type nodes.
- The '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.

```
InternalNodeReorder(QueryNode qnode)
 // Process the nodes which 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 4.2: Internal node reordering algorithm

- The ‘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.
- The ‘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.
- The ‘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.

The query tree is restructured using the above rules because the nodes that are being placed to left found out to be more selective in the experiments. The *gvcount* function in the algorithm 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))
```

In the query tree, 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.

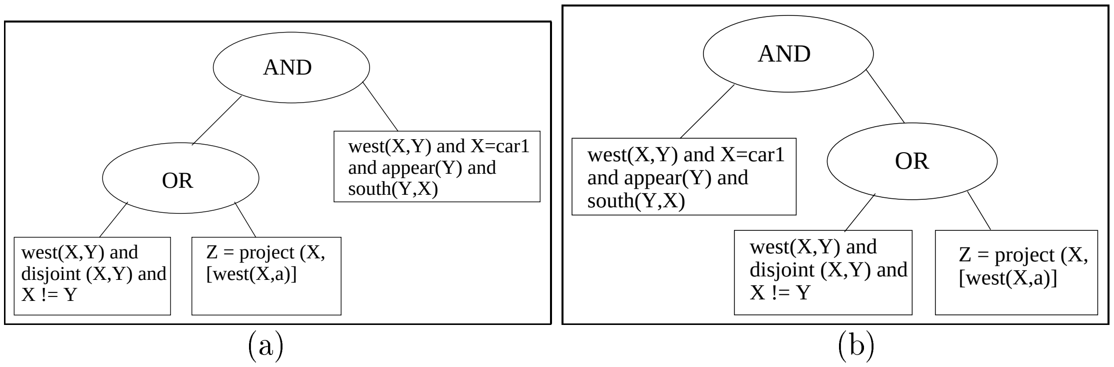
AND
OR
west(X,Y) and X=car1 and appear(Y) and south(Y,X)
west(X,Y) and disjoint (X,Y) and X != Y
Z = project (X, [west(X,a)]
(a)
AND
west(X,Y) and X=car1 and appear(Y) and south(Y,X)
OR
west(X,Y) and disjoint (X,Y) and X != Y
Z = project (X, [west(X,a)]
(b)

Figure 4.3: (a) Initial query tree for *Query 1* and (b) Query tree for *Query 1* after internal node reordering

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)])))
```

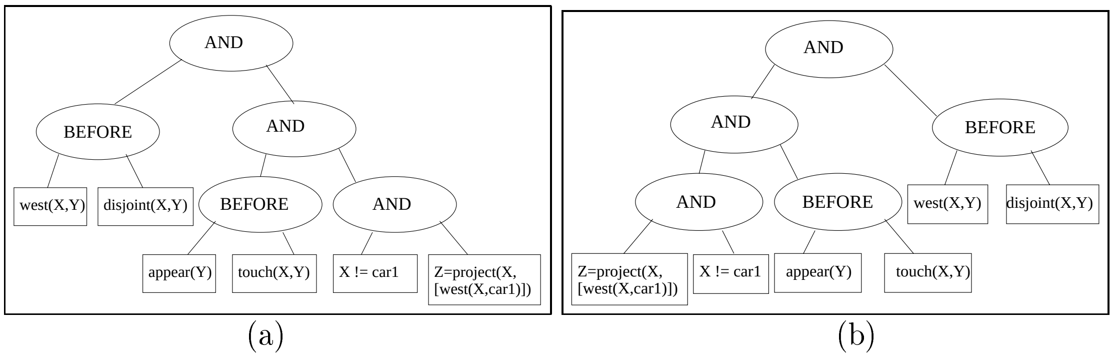

Figure 4.4: (a) Initial query tree for *Query 2* and (b) Query tree for *Query 2* after internal node reordering

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.

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. So 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 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. So 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 4.5.

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, the condition parts of the 'PRQ' type leaf nodes are replaced only. The functions used in the algorithm are explained in the sequel.

```
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 4.5: 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 4.6: The function that finds subquery tree of a leaf node

`FindPPQinList` function is used for locating the subquery tree of a particular leaf node in the subquery list (see Figure 4.6).

The `reorderAlg` function iterates on the subquery tree which is located in the subquery tree list and restructures this query tree (see Figure 4.7). This algorithm first locates the highest 'AND' type node in the subquery tree, if this node has left and right children and the left child is 'NOT-OR' typed and the right one is 'AND' typed, it exchanges the left and right nodes. If children are 'PPQ' or 'AND' typed 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 children are 'PPQ' or 'AND' typed and there is at least one 'NOT-OR' type node below these children, the algorithm finds out if the right child is a *maximal AND subtree* or not. If it is a *maximal AND subtree* then it exchanges the 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' typed (see Figure 4.8).

`OrderLeafNodes` function orders a *maximal AND subtree.* It first puts the leaf nodes into an array structure, sorts the array according to the fact base statistics and puts the leaf nodes back to the tree (see Figure 4.9).

`GetLeafNodes` function gets 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 4.10).

`SortLeafNodes` function sorts the leaf nodes according to the fact base statistics. It orders the relations in the increasing number of statistics (see Figure 4.11). 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.

```
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 4.7: 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 4.8: The function that finds if there is a 'NOT-OR' type node in a tree

```
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 4.9: 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 4.10: 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 4.11: The function that sorts leaf nodes

`PutLeafNodes` function puts the elements of an array structure to the leaf nodes of a tree. So, the nodes of the unsorted tree are replaced with the sorted nodes. (see Figure 4.12)

4.3.1 Examples

Some query examples 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 assuming that (*south facts* < *samelevel facts* < *west facts* < *overlap facts* < *disjoint facts* < *appear facts*) in the fact base.

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 4.12: The function that puts the elements to the leaf nodes

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

Query 1 after leaf node reordering:

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

Initial subquery tree for *Query 1* and subquery tree for *Query 1* after leaf node reordering, which are located in the subquery tree list, are shown in Figure 4.13.

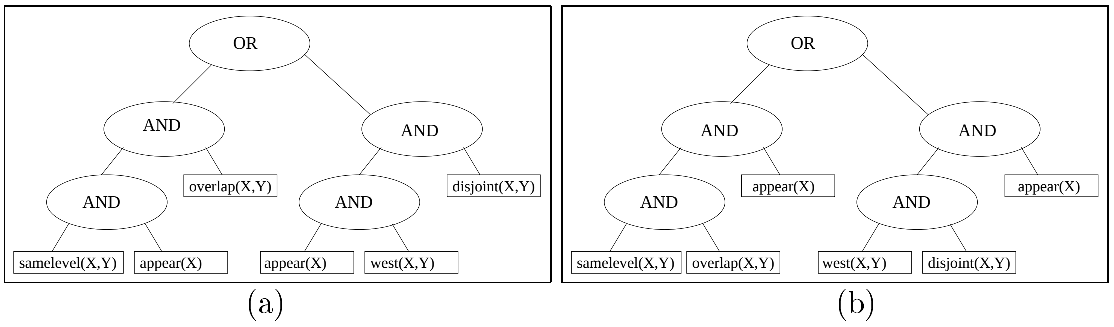

Figure 4.13: (a) Initial subquery tree for *Query 1* and (b) Subquery tree for *Query 1* after leaf node reordering

The relations in Query 2 are reordered in the second query, since *samelevel facts* < *overlap facts* < *appear facts* and *west facts* < *disjoint 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 relations in Query 1 are reordered as it can be seen from the second query, since *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.

Chapter 5

Performance Results

In this chapter, the performance results obtained for the query optimization algorithm are presented. Performance tests have been conducted on an example video that is extracted from television news. Performance tests have been carried out on Linux environment using the query processor of BilVideo implemented in C++. The performance parameters that affect query optimization are as follows:

1. *Size of the query:* While the size of the query is being increased, the performance gain obtained by our strategy also increases. For small sized queries, there will be small number of reorderings between the nodes, so the performance gain will be less compared to that with the large sized queries.

2. *Size of the video:* Size of the video is another parameter affecting query optimization since the size of the fact base is directly related with the size of the video. The performance gain will increase if the size of the video increases.

The organization of this chapter is as follows: In Section 5.1 statistics of the fact base used are provided. Example facts from this fact base can be found in Appendix A. The performance test results are presented Section 5.2. Example queries used in the experiments are discussed in Section 5.3.

5.1 Fact Base Statistics

The fact base of the example video is created using the fact extractor tool of BilVideo. The statistics of the video are given in Table 5.1. These statistics are used in the optimization algorithm to reorder the leaf nodes.

Table 5.1: 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

5.2 Performance Results

Five query sets are used in the performance tests. The first query set is used for testing the *Leaf Node Reordering* algorithm. The second set is used for testing the whole optimization algorithm. The third and fourth sets are constructed for testing the algorithm on different reorderings of the same query. Finally, the fifth set is used for testing the same query on different sizes of fact bases and result sets. The query sets can be found in Appendix B. *Optimization overhead* given in the results specifies the time that the optimization process takes and *performance* *gain* is defined in Formula 5.1. The first set of results are given in Table 5.2.

$$performance\ gain = \frac{(proc.\ time\ w/o\ opt. - proc.\ time\ with\ opt.)}{proc.\ time\ w/o\ opt.}. \tag{5.1}$$

Table 5.2: 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

These results show that leaf node reordering algorithm increases the performance of the query processor. There are different performance gains for each query in the set. This is because the performance gain depends on the size of the query and the difference between the initial query tree and the optimal query tree. The sizes of the first, ninth, eleventh and twelfth queries are small so 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 nodes are ordered starting from the relation with the smallest size of output to the relations with larger sized outputs. So 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 5.3.

Table 5.3: 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

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

The query optimization algorithm reduces the processing cost, because the subqueries with larger selectivities are processed before the subqueries with smaller selectivities. For example, if 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.

As it is mentioned previously, performance gain depends on the size and complexity of the query. Another factor affecting the performance is the difference between the initial query tree and the optimal query tree. The third and fourth performance tests are done 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 5.4. The fourth result set that uses a larger query tree is given in Table 5.5.

Table 5.4: 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

Table 5.5: 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

These two result sets show that when the query tree converges to the optimal query tree, the performance gain of the optimization algorithm decreases. This also justifies the correctness of the optimization algorithm.

The last performance test is done for investigating the effect of the query result set size on performance gain. A query is selected and its result set size is decreased by decreasing the fact base size at each step. The results of this test are given in Table 5.6.

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.

Table 5.6: 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

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 in multimedia 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.

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)]))
```

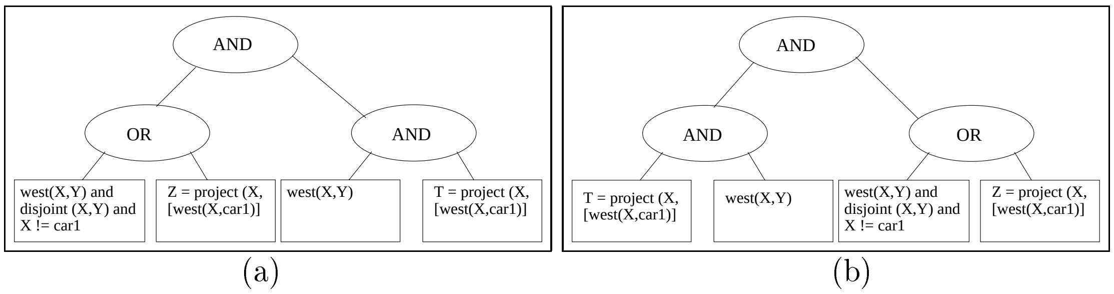
AND
OR
AND
west(X,Y) and disjoint (X,Y) and X != car1
Z = project (X, [west(X,car1)]
west(X,Y)
T = project (X, [west(X,car1)]
(a)
AND
AND
OR
T = project (X, [west(X,car1)]
west(X,Y)
west(X,Y) and disjoint (X,Y) and X != car1
Z = project (X, [west(X,car1)]
(b)

Figure 5.1: (a) Initial query tree for *Query 1* and (b) Query tree for *Query 1* after optimization

The initial query tree of Query 1 (Figure 5.1 (a)) is processed in 985 milliseconds and the optimized query tree (Figure 5.1 (b)) is processed in 286 milliseconds. So, the performance gain is 71%.

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]]))
```

The initial query tree of Query 2 (Figure 5.2 (a)) is processed in 845 milliseconds and the optimized query tree (Figure 5.2 (b)) is processed in 283 milliseconds. So, the performance gain is 67%.

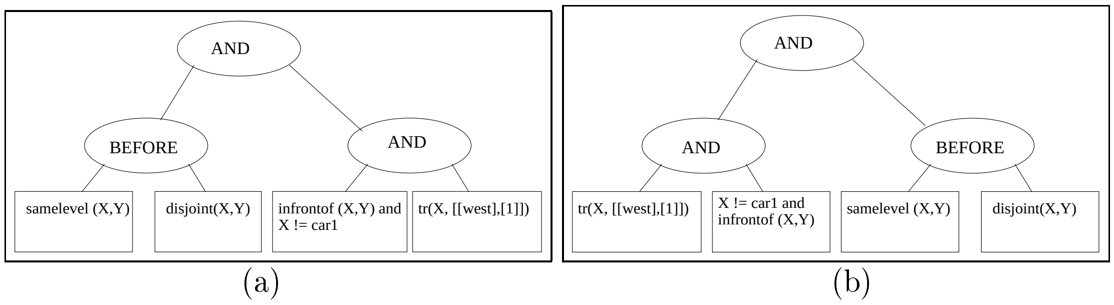

Figure 5.2: (a) Initial query tree for *Query 2* and (b) Query tree for *Query 2* after optimization

Chapter 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. Also, new database management systems such as multimedia databases require new techniques for query processing and query optimization.

In this thesis, we have presented a query optimization method for video database systems, which was implemented on a particular system, BilVideo. The proposed optimization method has two 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 justifies 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 which 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.

Bibliography

[1] M. E. Dönderler, Ö. Ulusoy, U. Güdükbay, A Rule-based Approach to Represent Spatio-Temporal Relations in Video Data, International Conference on Advances in Information Systems (ADVIS'2000), Izmir, Turkey, Lecture Notes in Computer Science (Springer Verlag), eds. T. Yakhno, vol. 1909, pp. 409-418, October 2000.

[2] M. E. Dönderler, Ö. Ulusoy, U. Güdükbay, A Rule-Based Video Database System Architecture, Information Sciences, vol. 143, no.1-4, pp. 13-45, 2002.

[3] M. E. Dönderler, Ö. Ulusoy, U. Güdükbay, Rule-Based Spatio-Temporal Query Processing for Video Databases (Submitted to the VLDB journal).

[4] N.S. Chang, K.S. Fu. Query by pictorial example. IEEE Transactions on Software Engineering, SE6, vol. 6, pp. 519-524, 1980.

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

[6] S. Chang, W. Chen, H. J. Meng, H. Sundaram, and D. Zhong. VideoQ: An automated content-based video search system using visual cues. In *Proc. of ACM Multimedia*, pp. 313-324, Seattle, Washington, USA, 1997.

[7] W. W. Chu, A. F. Cardenas, and R. K. Taira. A knowledge-based multimedia medical distributed database system - KMED. *Information Systems*, vol. 20, no. 2, pp. 75-96, 1995.

[8] E. Oomoto and K. Tanaka. Semantic Query Optimization for Tree and Chain Queries. *IEEE Transactions on Knowledge and Data Engineeering*, vol. 6, no. 1, February 1994.

[9] W. Sun and C. T. Yu. OVID: Design and implementation of a video object database system. *IEEE Transactions on Knowledge and Data Engineeering*, vol. 5, pp. 629-643, 1993.

[10] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, vol. 16, no. 2, pp. 111-152, June 1984.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, vol. 25, no. 2, pp. 73-170, June 1993.

[12] G. Menekse, F. Polat, A. Cosar. Alternative Plan Generation Methods for Multiple Query Optimization, *ISCIS'98*, *Advances in Computer and Information Sciences'98*, eds. U. Güdükbay et al., pp. 246-253, 1998.

[13] S. Chaudhuri. An Overview of Query Optimization in Relational Systems, In *Proc. of Principles of Database Systems'98*, pp. 34-43, 1998.

[14] A. Soffer, H. Samet. Query Processsing and Optimization for Pictorial Query Trees, *Visual Information and Information Systems - VISUAL99* (D. P. Huijsmans and A. W. M. Smeulders, Eds.), Lecture Notes in Computer Science 1614, Springer, Berlin, 1999, pp. 60-67.

[15] L. P. Mahalingam, K. S. Candan. Query Optimization in the Presence of Top-k Predicates, *Multimedia Information Systems 2001*, pp. 31-40.

[16] R. H. Guting, M. H. Bohlen, M. Ervig, C. S. Jensen, N. A. Lorentzos, M. Schneider, M. Vazirgiannis. A foundation for representing and querying moving objects, *ACM Transactions on Database Systems*, vol. 25, no. 1, pp. 1-42, 2000.

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

[18] M. Nabil, A. H. Ngu, J. Shepherd. Modelig and retrieval of moving objects, *Multimedia Tools and Applications*, vol. 13, pp. 35-71, 2001.

[19] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and querying moving objects, In *Proc. of IEEE Data Engineering*, pp. 422-432, 1997.

Appendix A

Sample Fact Base for an Example Video

This is an example video containing 16,351 frames and 98 salient objects. Some salient objects in the video are `tank1`, `car1`, `bodyguard1` and `powell`. The video is extracted from television news.

```
// Directional Relations
west(tank1,car1,259).
west(car1,car2,259).
west(tank1,car1,272).
west(car1,car2,272).
west(car2,car3,272).
west(tank1,car1,277).
west(car1,car2,277).
west(car2,car3,277).
west(car3,car4,277).
west(tank1,car1,280).
west(car1,car2,280).
west(car2,car3,280).
west(car3,car4,280).
west(car4,car5,280).
west(tank1,car1,282).
west(car1,car2,282).
west(car2,car3,282).
west(car3,car4,282).
west(car4,car5,282).
west(car1,car2,287).
west(car2,car3,287).
west(car3,car4,287).
west(car4,car5,287).
west(car1,car2,298).
west(car2,car3,298).
west(car3,car4,298).
west(car4,car5,298).
west(car1,car2,303).
west(car2,car3,303).
west(car3,car4,303).
west(car4,car5,303).
west(tank3,tank4,366).
west(tank3,tank4,386).
west(tank3,tank4,395).
west(tank3,tank4,408).
west(tank5,israelisoldier1,409).
west(tank5,israelisoldier1,435).
west(tank5,israelisoldier1,456).
west(palestinianofficer4,bodyguard2,484).
south(palestinianofficer1,officialcar,527).
south(palestinianofficer4,powell,527).
south(palestinianofficer3,officialcar,531).
south(palestinianofficer1,officialcar,531).
south(palestinianofficer4,powell,531).
south(palestinianofficer1,officialcar,535).
south(palestinianofficer4,powell,535).
south(palestinianofficer1,officialcar,542).
south(palestinianofficer4,powell,542).
south(palestinianofficer3,bodyguard1,542).
south(palestinianofficer1,officialcar,547).
south(palestinianofficer4,powell,547).
south(palestinianofficer3,bodyguard1,547).
south(palestinianofficer1,officialcar,550).
south(palestinianofficer4,powell,550).
south(palestinianofficer3,bodyguard1,550).
south(palestinianofficer4,officialcar,560).
south(palestinianofficer3,bodyguard1,560).
south(palestinianofficer4,palestinianofficer1,568).
south(palestinianofficer4,officialcar,568).
south(palestinianofficer3,bodyguard1,568).
south(palestinianofficer4,officialcar,572).
south(palestinianofficer3,bodyguard1,572).
south(palestinianofficer4,officialcar,578).
south(palestinianofficer3,bodyguard1,578).
south(palestinianofficer4,officialcar,579).

//Topological Relations
disjoint(car2,car5,303).
disjoint(car1,car5,303).
disjoint(tank5,israelisoldier1,409).
disjoint(tank5,israelisoldier1,435).
disjoint(tank5,israelisoldier1,456).
disjoint(palestinianofficer2,bodyguard1,484).
disjoint(palestinianofficer3,powell,484).
disjoint(powell,bodyguard2,484).
disjoint(palestinianofficer2,palestinianofficer4,484).
disjoint(palestinianofficer4,bodyguard2,484).
disjoint(palestinianofficer2,palestinianofficer1,484).
disjoint(bodyguard1,powell,484).
disjoint(bodyguard1,bodyguard2,484).
disjoint(palestinianofficer2,bodyguard2,484).
disjoint(palestinianofficer2,powell,484).
disjoint(palestinianofficer1,bodyguard2,484).
disjoint(palestinianofficer3,bodyguard2,484).
disjoint(palestinianofficer2,bodyguard1,492).
disjoint(palestinianofficer3,powell,492).
disjoint(palestinianofficer3,palestinianofficer4,492).
disjoint(bodyguard1,palestinianofficer4,492).
disjoint(powell,bodyguard2,492).
disjoint(palestinianofficer2,palestinianofficer4,492).
disjoint(palestinianofficer4,bodyguard2,492).
disjoint(palestinianofficer2,palestinianofficer1,492).
disjoint(bodyguard1,powell,492).
disjoint(bodyguard1,bodyguard2,492).
disjoint(palestinianofficer2,bodyguard2,492).
disjoint(palestinianofficer2,powell,492).
disjoint(palestinianofficer1,bodyguard2,492).
disjoint(palestinianofficer3,bodyguard2,492).
disjoint(palestinianofficer3,powell,498).
disjoint(palestinianofficer3,palestinianofficer4,498).
disjoint(bodyguard1,palestinianofficer4,498).
disjoint(powell,bodyguard2,498).
disjoint(palestinianofficer2,palestinianofficer4,498).
overlap(palestinianofficer4,powell,503).
overlap(palestinianofficer1,powell,503).
overlap(palestinianofficer4,palestinianofficer1,503).
overlap(palestinianofficer4,officialcar,503).
overlap(palestinianofficer3,palestinianofficer1,503).
overlap(palestinianofficer3,bodyguard1,503).
overlap(palestinianofficer3,officialcar,503).
overlap(palestinianofficer2,officialcar,503).
overlap(palestinianofficer1,officialcar,503).
overlap(officialcar,powell,503).
overlap(bodyguard1,officialcar,503).
overlap(palestinianofficer1,bodyguard1,503).
overlap(bodyguard2,officialcar,503).
overlap(palestinianofficer2,bodyguard1,512).
overlap(palestinianofficer3,powell,512).
overlap(palestinianofficer4,powell,512).
overlap(palestinianofficer1,powell,512).
overlap(palestinianofficer4,palestinianofficer1,512).
overlap(palestinianofficer4,officialcar,512).
overlap(palestinianofficer3,palestinianofficer1,512).
overlap(palestinianofficer3,bodyguard1,512).

// 3-D Relations
touchfrombehind(handeataizi,erbil,14079).
touchfrombehind(handeataizi,erbil,14084).
touchfrombehind(handeataizi,erbil,14094).
touchfrombehind(erbil,prizecheck,14434).
touchfrombehind(erbil,prizecheck,14469).
touchfrombehind(erbil,prizecheck,14474).
touchfrombehind(erbil,prizecheck,14494).
touchfrombehind(erbil,prizecheck,14499).
touchfrombehind(erbil,prizecheck,14569).
strictlyinfrontof(tank5,israelisoldier1,476).
strictlyinfrontof(tank5,israelisoldier1,483).
strictlyinfrontof(powell,officialcar,484).
strictlyinfrontof(powell,officialcar,492).
strictlyinfrontof(powell,officialcar,498).
strictlyinfrontof(powell,officialcar,503).
strictlyinfrontof(powell,officialcar,512).
strictlyinfrontof(powell,officialcar,518).
strictlyinfrontof(powell,officialcar,520).
strictlyinfrontof(powell,officialcar,527).
strictlyinfrontof(powell,officialcar,531).
strictlyinfrontof(powell,officialcar,535).
strictlyinfrontof(powell,officialcar,542).
infrontof(policevehicle2,tank10,3076).
infrontof(vuralsavas,ozbek,10491).
infrontof(vuralsavas,ozbek,11041).
infrontof(vuralsavas,ozbek,11476).
infrontof(ecevit,ismailcem,12591).
infrontof(erbil,cemyilmaz,13689).
infrontof(erbil,cemyilmaz,13694).
infrontof(erbil,cenkeren,14264).
infrontof(erbil,cenkeren,14269).
infrontof(cemyilmaz,cenkeren,14304).
infrontof(cemyilmaz,cenkeren,14309).
infrontof(cemyilmaz,cenkeren,14324).
infrontof(yelizyesilmen,erbil,15649).
samelevel(car1,car2,259).
samelevel(tank1,car3,272).
samelevel(car1,car2,272).
samelevel(tank1,car3,277).
samelevel(car1,car2,277).
samelevel(car3,car4,277).
samelevel(tank1,car3,280).
samelevel(car1,car2,280).
samelevel(car3,car4,280).
samelevel(tank1,car3,282).
samelevel(car1,car2,282).
samelevel(car3,car4,282).
samelevel(car3,car5,282).
samelevel(car1,car2,287).
samelevel(car3,car4,287).
samelevel(car3,car5,287).
samelevel(car1,car2,298).

// Trajectory Facts
tr(car3,[west,north],[177,5],[[272,298],[298,301]]).
tr(car4,[west],[158],[[277,308]]).
tr(car5,[west,northwest],[116,8],[[280,303],[303,308]]).
tr(tank4,[west],[12],[[366,386]]).
tr(tank3,[east],[12],[[386,395]]).

// Appear Facts
appear(tank1,[[259,286]]).
appear(car1,[[259,365]]).
appear(car2,[[259,365]]).
appear(car3,[[272,365]]).
appear(car4,[[277,365]]).
appear(car5,[[280,365]]).
appear(tank2,[[366,408]]).
appear(tank3,[[366,408]]).
appear(tank4,[[366,408]]).
appear(tank5,[[409,483]]).
appear(israelisoldier1,[[409,483]]).
appear(palestinianofficer4,[[484,606]]).
appear(palestinianofficer2,[[484,698]]).
```

Appendix B

Query Sets

Query set that is 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)])
```

Query set that is 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]]))
```

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)
   ```

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)]))
```