University of Alberta

Multimedia Extensions To Database Query Languages

by

John Z. Li, M. Tamer Özsu, Duane Szafron
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
{zhong,ozsu,duane}@cs.ualberta.ca

Technical Report TR 97–01
January 1997

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Multimedia Extensions To Database Query Languages *

Abstract

Declarative query languages are an important feature of database management systems and have played an important role in their success. As database management technology enters the multimedia information system domain, the availability of query languages for multimedia applications will be equally important. However, one common problem with currently existing multimedia query languages is their lack of generality. They are designed either for a certain medium (e.g. images) or special applications (e.g., medical, geographical information systems). We describe general multimedia queries based on the ODMG's Object Query Language (OQL) and TIGUKAT Query Language (TQL). In order to capture the temporal and spatial relationships in multimedia data, both OQL and TQL are extended by a set of multimedia primitives. These extended OQL and TQL also include functions for query presentation. We illustrate the extended language features by query examples.

Keywords: multimedia, OQL, TQL, MOQL, MTQL, Object-oriented, database, query, language

*This research is supported by a grant from the Canadian Institute for Telecommunications Research (CITR) under the Network of Centres of Excellence (NCE) program of the Government of Canada.

Contents

1 Introduction **5**

2 Related Work **8**

3 Query Languages and The Extensions **10**

3.1 Object Query Language 10

3.2 TIGUKAT Query Language 13

4 Spatial Primitives **15**

4.1 Spatial Predicates 16

4.2 Spatial Functions 19

5 Temporal Primitives **20**

5.1 Temporal Functions 21

5.2 Continuous Media Functions 22

5.3 Presentation Functions 25

6 Implementation of The Languages **29**

7 Conclusion **30**

A Appendix: MOQL Specification **35**

A.1 Axiom 35

A.2 Basic 35

A.3 Simple Expression 35

A.4 Comparison 35

A.5 Boolean Expression 35

A.6 Constructor 35

A.7 Accessor 36

A.8 Collection Expression 36

A.9 Select Expression 36

A.10 Set Expression 36

A.11 Conversion 37

A.12 Spatial Expression 37

A.13 Temporall Expression 38
A.14 Boolean Expression 38
A.15 Presentation Layout 38

B Appendix: MTQL Specification 40
B.1 Basic Statements 40
B.2 MTQL Terms 40
B.3 Spatial Expression 41
B.4 Temporall Expression 41
B.5 Boolean Expression 42
B.6 Presentation Layout 42

List of Figures

1 Definitions of Topological Relations 18
2 Query Processing and Query Tree 44

List of Tables

1 Spatial Predicates 17
2 Spatial Functions 20
3 Continuous Media Functions 22

1 Introduction

Multimedia query languages are important, since query languages are an integral feature of database management systems (DBMS). One of the basic functionalities of a DBMS is to be able to efficiently process declarative user queries. The penetration of DBMS technology into multimedia information systems necessitates the development of query languages appropriate for this domain. The complex spatial and temporal relationships inherent in the wide range of multimedia data types make a multimedia query language quite different from its counterpart in traditional DBMSs. For example, the query languages of traditional DBMSs only deal with exact-match queries on conventional data types. Although this might be sufficient to deal with queries posed against metadata or annotations of multimedia data, content-based information retrieval requires non-exact-match (fuzzy) queries which go beyond the traditional approaches.

A powerful query language significantly helps users to manipulate a multimedia database. It also helps to maintain the desired independence between the database and the application. Effective query languages must be user friendly for both naive and expert users. Furthermore, *query presentation*, which refers to the way query results are presented, is more complex in multimedia systems than traditional DBMSs. This is because multimedia presentations have to take into account the synchronization of various media. In recent years, there have been many multimedia query language proposals [BRG91, OM88, RFS88, AB91, DG92, CIT+93, Ege94, Güt94, CIT94, HK95, KC96, ATS96, MS96, GR96, MHM96]. These proposals can be classified into three categories:

- Those that are proposed as a new language: [CIT+93, HK95, KC96, ATS96, GR96].
- Languages that are based on a logic or a functional programming approach: [DG92, MS96].
- Languages that are extensions of SQL: [BRG91, OM88, RFS88, AB91, OT93, Ege94, Güt94, CIT94, MHM96].

One problem with a brand new multimedia query language is the lack of acceptance by users. In general, it is difficult to convince users to learn and use a new language

for each application. Another problem with the current proposals is the lack of theoretical results about the soundness and expressive power of new languages. In fact, none of the proposed new languages has ever addressed this problem.

Specifying queries using logic and functional programming approaches is relatively difficult. Therefore, this method is not very attractive, despite the expressive power of these languages. They may be suitable as lower level multimedia query languages, but not as user languages.

The majority of existing approaches to designing multimedia query languages are based on extensions of SQL. This is generally due to the popularity of SQL for traditional database applications. A common problem with all the existing SQL-based multimedia query languages is that they are designed either for a particular medium or for a specific application domain, not for general use. For example, VideoSQL [OT93] is used only for video databases, SEQL [CIT94] is designed mainly for medical sequence image databases, ESQL [AB91] is good only for image databases, and PSQL [RFS88] and SpatialSQL [Ege94] are suitable only for spatial databases.

Are there any general query languages for multimedia databases? How can they be formally defined so that they are independent of particular media and specific applications? These are the question that this paper addresses. It is well-known that object-oriented technology is a promising one for dealing with multimedia data. As a result, almost all multimedia DBMSs are directly or indirectly (by extending relational models into object-oriented models) based on object-oriented technology. Object Query Language (OQL) [Cat94] has been proposed by Object Database Management Group (ODMG) which is a consortium of object database management system (ODBMS) vendors and interested parties working on standards to allow portability of customer software across ODBMS products. Syntactically, OQL is very similar to SQL. It is currently supported by most major ODBMS vendors and its popularity should increase as the ODBMS market grows. To the best of our knowledge, no multimedia extensions to OQL exist.

We also consider another general object database query language, TIGUKAT [PLÖS93]. TIGUKAT has an extensible, uniform, behavioral query model and query language. The TIGUKAT model is purely behavioral in nature, supports full encapsulation of objects, defines a clear separation between primitive components such as types, classes, collections, behaviors, functions, etc., and incorporates a uniform

semantics over objects. Queries are modeled as type and behavior extensions to the base object model, thus incorporating queries as an extensible part of the model itself. TIGUKAT Query Language (TQL) is an SQL-like ad hoc query language for database users.

In this report, we propose multimedia extensions to OQL and TQL to develop a general-purpose query language for multimedia DBMSs. The extended languages are called Multimedia Object Query Language (MOQL) and Multimedia TIGUKAT Query Language (MTQL) respectively. These include extensions of spatial properties, temporal properties, and presentation properties to OQL and TQL. These extensions are introduced through predicates and functions. The major contribution of this work are:

- a complete multimedia query language which supports general media and applications;
- definition of spatial and temporal operators which support content-based queries; and
- a multimedia OQL preprocessor which parses MOQL queries and generates algebraic trees for subsequent optimization.

All the multimedia query examples in this report will be illustrated by both MOQL and MTQL.

The major differences between TQL and OQL are characterized as follows:

- Functions are allowed in OQL, but not in TQL.
- OQL supports aggregate functions such **average**, **sum**, **group by**, etc. while TQL does not.
- OQL allows object references (pointers in C++ context) which are not addressed in TQL.
- An OQL query can be in any form while a TQL query has to use **select** clause.

In general, OQL is more expressive than TQL.

Visual query languages have also received attention as interfaces to multimedia databases. Visual query systems are especially good for novice or casual users. A user

can manipulate the database easily, with only limited knowledge of the underlying system. For experienced users, visual queries can still be beneficial in those cases where queries are difficult to express syntactically. Even though we do not specifically address visual query languages in this report, MOQL and MTQL establish the basis for these interfaces. Such an interface based on MOQL is currently under development.

The rest of the report is organized as follows. Section 2 reviews the related work in multimedia query languages. Section 3 introduces ODMG's OQL and TIGUKAT's TQL, as well as our basic extensions. Section 4 introduces our extensions to spatial primitives which describe relationships between spatial objects. We define a set of spatial functions and predicates to support complex spatial operations. Section 5 introduces our extensions to temporal primitives, as well as the query presentation functions. We focus on video data in this section by discussing the video functions in some depth. Section 6 briefly discusses the current status of implementing MOQL as part of a full fledged multimedia DBMS. Section 7 summarizes our work and discusses possible future work.

2 Related Work

PSQL (Pictorial SQL) [RFS88] is designed for pictorial databases which require efficient and direct spatial search, based on the geometric form of spatial objects and relationships. It allows a user to directly manipulate spatial objects. An important feature of PSQL is the introduction of many spatial operators, such as *nearest* and *furthest* for point objects, *intersect* and *not-intersect* for segment objects, and *cover*, and *overlap* for region objects. Syntactically, there is not much difference from the standard SQL.

EVA [DG92] is, an object-oriented language, based on functional language features with roots in conventional set theory. It is formally defined using the mathematical framework of a many sorted algebra. Although EVA has defined a set of spatio-temporal operators to support query presentation, it lacks some useful presentation features, such as changing display speeds and time constraints (e.g., presenting some object for 20 minutes). Furthermore, EVA does not support spatial queries or video data.

A knowledge-based object-oriented query language, called PICQUERY$^+$, is proposed in [CIT$^+$93]. PICQUERY$^+$ is a high-level domain-independent query language designed for image and alphanumeric database management. It allows users to specify conventional arithmetic queries as well as evolutionary and temporal queries. The main PICQUERY$^+$ operations include panning, rotating, zooming, superimposing, color transforming, edge detecting, similarity retrieving, segmenting, and geometric operations. A template technique has been used in PICQUERY$^+$ to facilitate user queries. Such *query templates* are used in PICQUERY$^+$ to specify predicates to constrain the database view.

SEQL (Spatial Evolutionary Query Language) [CIT94], a direct extension of SQL, is proposed to operate on the spatial evolutionary domains of medical images. In addition to alphanumeric predicates, SEQL contains constructs to specify spatial, temporal, and evolutionary conditions. A **when** clause is added to the language, which selects the appropriate snapshot of the data of interest at a particular point in time. It supports temporal functions which manipulate time points (such as *start time*, *end time* etc.), temporal ordering of an object history (such as *first*, *last*, *next*, etc.), and temporal interval (such as *before*, *after*, *during*, etc.). Another extension is the addition of a **which** clause which describes various evolutionary processes on a set of evolving objects. Unfortunately, SEQL supports only image databases.

Marcus and Subrahmanian [MS96] have proposed a formal theoretical framework for characterizing multimedia information systems. The framework includes a logical query language that integrates diverse media. This is a first attempt at mathematically characterizing multimedia database systems. The model is independent of any specific application domain and provides the possibility of uniformly incorporating both query languages and access methods, based on multimedia index structures. This model defines a special data structure, called a *frame*, which is used for data access. The query language is based on logic programming and it makes extensive use of predicates and functions. Such a query language is suitable as an intermediate query language between a higher level language (such as OQL) and a lower level language (such as an object algebra).

Two new query languages, MMQL (Multimedia Query Language) and CVQL (Content-based Video Query Language), for video databases are described in [KC96] and [ATS96] respectively. A major problem with MMQL is that it does not support

spatial queries which is fundamental to a multimedia query language in our opinion. CVQL is defined based on vide frame-sequences. Therefore, to query a video database using CVQL, a user must have good knowledge about the video (or frame sequence) on which he/she intends to query. ESQL [AB91] is an image domain query language for the relational model. The query language in [BRG91] is designed for multimedia office documents. Some research [HK95, PS95] has also been done in supporting multimedia content specification and retrieval in the design of a multimedia query language.

3 Query Languages and The Extensions

3.1 Object Query Language

OQL defines an orthogonal expression language, in the sense that all operators can be composed with each other as long as the types of the operands are correct. It deals with complex objects without changing the set construct and the select-from-where clause. It is close to SQL 92 with object-oriented extensions such as complex objects, object identity, path expressions, polymorphism, operation invocation, and late binding. It includes high-level primitives to deal with bulk objects like structures, lists, and arrays. As a stand-alone language, OQL allows users to query objects by using their names as entry points into a database. As an embedded language, OQL allows applications to query objects that are supported by the native programming language, using expressions that yield atoms, structures, collections, and literals. An OQL query is a function which returns an object whose type may be inferred from the operators contributing to the query expression. OQL has one basic statement for retrieving information:

```
  select [ distinct ] projection_attributes
  from  query [ [ as ] identifier ] {, query [ [ as ] identifier ] }
 [ where  query 1]
 [ group by  partition_attributes ]
 [ having  query ]
 [ order by  sort_criterion {, sort_criterion }]
```

where projection_attributes is a list of attribute names whose values are to be re-

[1] *query* is not a proper name here and a better name might be *predicate-expression*. However, since this the standard OQL term we do not change it.

trieved by the query. In the **from** clause, a variable has to be bound to a set of objects, an extent, or a query. In the **where** clause, the query is a conditional search expression that identifies the objects to be retrieved by the query. The conditional search expression can be any OQL query. In OQL, *query* is very general as described in Appendix A. The clauses **group by**, **order by**, and **having** have the same semantics as their counterparts in SQL.

Query examples will be introduced to describe different features of OQL and MOQL. In the interest of saving space, we do not define the schema of the database against which these queries are specified. The queries should be self-explanatory. We discuss those aspects of the queries which require explanation. We follow ODMG's conventions: a class name has its first character capitalized, a class extent is represented by the class name with its plural form (E.g. Employee is the name of class Employee and Employees is its extent), and an object is identified by a special font (E.g. object x is denoted by x). We now illustrate the basic statements by using some example queries.

Query 1 Retrieve the birth date of the employee whose name is *John*:

```
select   e.birthDate
from     Employees e
where    e.name="John"
```

This query involves only one extent, *Employees*, listed in the **from** clause. The query selects all the employees from class *Employee* that satisfy the condition of the **where** clause, then projects the result on the *birthDate* attribute. The result is a set of birth dates.

Query 2 Return a set of structures consisting of the ages and salaries of employees, whose names are "John" and whose seniorities are greater than 20:

```
select   struct(a:e.age, s:e.salary)
from     (select   f
          from     Employees f
          where    f.seniority > 20)
          as e
where    e.name = "John"
```

This query shows that the **from** clause does not have to be a class extent; it may contain queries too. The result of this query is a literal of the type set<struct>, namely, set<struct(a: int, s: float)>.

Query 3 Retrieve the street addresses of the spouses of employees who live in Paris:

```
select   e.spouse.address.street
from     Employees e
where    e.lives_in("Paris")
```

The **select** clause includes a *path expression* and a method invocation. A method can return a complex object or a collection that can be embedded in a complex path expression. If *spouse* is a method defined on the class *Employee* which returns an object of class *Person*, the result of the above query is the set of their spouses' street names for those employees who lives in Paris. Although *spouse* is a method we traverse it as if it were a relationship. Moreover, the where clause contains a method, *lives_in*, which has one parameter.

Most of the extensions that we introduce to OQL are in the **where** clause. These extensions are in the form of three new predicate expressions: *spatial_expression*, *temporal_expression*, and *contain_predicate*. A full specification of these extensions is given in Appendix A. The *spatial_expression* is the spatial extension which includes spatial objects (such as points, lines, circles etc.), spatial functions (such as length, area, intersection, etc.), and spatial predicates (such as cover, disjoint, left etc.). A detailed discussion of spatial extensions is given in Section 4. The *temporal_expression* deals with temporal objects, temporal functions, and temporal predicates; these are discussed in Section 5. The *contain_predicate* has the basic form:

contain_predicate ::= media_object **contains** salientObject

where, *media_object* represents an instance of a particular medium type, e.g., an image object or a video object, while salientObject is a *salient object* which is defined as an interesting physical object in a media object. Each media object has many salient objects, e.g. persons, houses, cars, etc. The **contains** predicate checks whether a salient object is in a particular media object or not.

Query 4 Find images in which a person appears.

```
select   m
from     Images m, Persons p
where    m contains p
```

This simple query uses the **contains** predicate which checks whether a person p is in image m. The full set of multimedia extensions to OQL that we propose is specified in Appendix A. In the following sections, we discuss these extensions and give intuitive examples to demonstrate them.

3.2 TIGUKAT Query Language

The important characteristics of the TIGUKAT model are its *behaviorality* and and its *uniformity* [ÖPS+95]. The model is *behavioral* in the sense that all access and manipulation of objects is based on the application of behaviors to objects. The model is *uniform* in that every component of information, including its semantics, is modeled as a *first-class object* with well-defined behavior. The typical object-oriented features, such as strong object identity, abstract types, strong typing, complex objects, full encapsulation, multiple inheritance, and parametric types are also supported.

TIGUKAT Query Language (TQL) is an SQL-like ad hoc query language for database users to retrieve objects. TQL extends the SQL structure by accepting path expression. Thus, path expressions can be used in the *select clause* to navigate through the schema, in the *from clause* if the result of the application of behaviors is a finite collection, and in the *where clause* as predicates. The basic query statement of the TQL is the *select statement*. It operates on a set of input collections and it always returns a new collection as the result. The general syntax of the select statement is:

select object_variable_list
[**into** [**persistent** [**all**]] collection_reference
from range_variableList
where boolean_formula

where the **select** clause in this statement identifies objects which are to be returned in a new collection. There can be one or more object variables in this clause. They can be in form of simple variables, path expressions, index variables, or constants. The **into** clause declares a reference to a new collection returned as a result of a query. It is useful when the TQL is embedded in some other programming languages. In addition, the result collection can be made persistent by specifying it in the **into** clause. The **persistent** subclause makes only the container object persistent. The **from** clause declares ranges of object variables in the select and where clauses. Every object variable can range over either an existing collection, or a collection returned as a result of of a subquery, while a subquery can be either given explicitly, or as a reference to a query object. The TQL **where** clause is similar to the SQL **where** clause and the formal definitions of TQL can be found in Appendix B. Note the TQL does not support aggregate functions, such as **group by**, **order by**, or **having**.

Query examples will be introduced to describe different features of TQL and MTQL. Following TIGUKAT's convention, a reference prefixed by "T_" refers to

a type, "*C_*" to a class, "*B_*" to a behavior, and "T_X< T_Y >" to the type T_X parameterized by the type T_Y. For example, T_person refers to a type, *C_person* to its class, *B_age* to one of its behaviors and T_collection< T_person > to the type of collections of persons. A reference such as *David*, without a prefix, denotes some other application specific reference. Note that the model separates the definition of object characteristics (a *type*) from the mechanism for maintaining instances of a particular type (a *class*).

Query 5 Retrieve the birth date of the employee whose name is *John*:

```
select  e.B_birthDate
from    e in C_employee
where   e.B_name.B_equal("John")
```

This query is almost identical to **Query 1**.

Query 6 Return a set of ages and salaries of all employees whose names are "John" and seniorities are greater than 20:

```
select  e.B_age, e.B_salary
from    e in    (select  f
                 from    f in C_employee
                 where   f.B_seniority.B_greaterthan(20))
where   e.B_name.B_equal("John")
```

This query is similar to **Query 2**.

Query 7 Retrieve the street addresses of the spouses of employees who live in Paris:

```
select  e.B_spouse.B_address.B_street
from    e in C_employee
where   e.B_lives_in("Paris")
```

The **select** clause includes a *path expression* and a method invocation.

Similar to OQL, most of the extensions that we introduce to TQL are in the **where** clause in the form of three new predicate expressions: *spatial_expression*, *temporal_expression*, and *boolean_expression*. A full specification of these extensions is given in Appendix B. The *boolean_expression* has the basic form:

boolean_expression ::= contain_predicate | boolean_function

where contain_predicate and boolean_function are described in the previous subsection.

Query 8 Find images in which a person appears.

select *m*
from *m* **in** *C_image*, *p* **in** *C_person*
where *m* **contains** *p*

This simple query uses the **contains** predicate which checks whether a person *p* is in image *m*.

4 Spatial Primitives

Many applications depend on spatial relationships among data. *Spatial data* pertains to the space occupied by objects and includes points, lines, squares, regions, volumes, etc. The special requirements of multimedia query languages in supporting spatial relationships have been investigated. From a user's point of view, the following requirements are necessary for supporting spatial queries in a multimedia information system:

- Support should be provided for object domains which consist of *complex* (structured) spatial objects in addition to simple (unstructured) points and alphanumeric domains. These spatial objects must be accessible by pointing to them or describing the space they occupy, and not just by referencing their encodings.

- Support should exist for *direct spatial searches*, which locate the spatial objects in given areas of images. This can resolve queries of the form "*Find all the faces in the upper half of an image or a video frame*".

- It should be possible to perform *hybrid spatial searches*, which selects objects based on some attributes and some associations between attributes and spatial objects. This can resolve queries of the form "*Display the person's name, age, and an image in which the person is riding on a horse*".

- Support should exist for *complex spatial searches*, which locate spatial objects across the database by using set-theoretic operations over spatial attributes. This can resolve queries of the form "*Find all the roads which pass through city X*" where one may need to get the location coordinates of city X and then check road maps to see which ones contain the coordinates.

- Support should be provided to perform *direct spatial computations*, which compute specialized simple and aggregate functions from the images. This can resolve queries of the form "*Tell me the combined areas of all lakes in this image*".

- Finally, support should exist for *spatio-temporal queries* which involve not only spatial relations, but temporal relations as well. This can resolve queries of the form "*Find a clip in which a person is at the left of another person and later on the two exchange their positions*".

4.1 Spatial Predicates

A spatial predicate compares the spatial properties of spatial objects and returns a boolean value as the result. We define *picture* as any digital visual data which include images, maps, and computer generated graphics. We identify a set of general pictorial predicates. Note that since there are big differences between different types of pictures (such as a GIF image and a computer-generated polygon), more precise definitions are possible, but they would have to consider different types of pictures and different formats of the same type of pictures. This is beyond the focus of this paper. Instead, we define an image in an abstract format so we can provide generic functions on images.

An image is defined by a quadruple

$$< SalientSet, SpatialSet, ColorSet, TextualSet >$$

where

- $SalientSet$: the set of salient objects in the image;

- $SpatialSet$: the spatial properties of salient objects, such as shapes, locations etc.;

- $ColorSet$: the color properties of salient objects and image background;

- $TextualSet$: the textual values of salient objects and image background.

The set of pictorial predicates is:

- *identical*: check if two pictures are identical; two pictures are identical if their sets $SalientSet$, $SpatialSet$, $ColorSet$, and $TextualSet$ are equal respectively;
- *coincident*: check if two pictures have identical $SpatialSet$s;
- *subpicture*: check if one picture is contained inside another picture, which requires the picture's $SalientSet$, $SpatialSet$, $ColorSet$, and $TextualSet$ be the subsets of another picture's respectively;
- *similar*: check if two pictures are similar with respect to some metrics; such metrics can be salient objects, spatial relationships, colors, textures or combination of these;
- *contains*: check if a picture contains a particular salient object; i.e. check if a salient object is an element of $SalientSet$.

We define only three spatial primitives: *point*, *line*, and *region*. Although other constructors, such as circle, rectangle etc., are provided, they are all special cases of *region*. A region may be represented by a set of points, a set of lines, a set of polygons, or other forms (e.g. a point and a radius) in some universe. In the case of continuous two-dimensional space, the universe is the whole plane. In the case of continuous three-dimensional space, the universe is the whole three-dimensional space. Regions in these cases correspond to areas and volumes respectively. Table 1 shows basic spatial predicates defined in MOQL.

	point	line	region
point	nearest, farthest	within, midpoint	centroid, inside
line	cross	intersect	inside, cross
region	cover	cover, cross	topological_predicate, directional_predicate

Table 1: Spatial Predicates

The operands of the spatial predicates must be the same or compatible object types. For example, predicates *nearest* and *farthest* can apply only to two point

objects, predicates *within* and *midpoint* can apply only to a point and a line, and predicate *cover* may apply to a region and a point or to a region and a line. We do not give exact definitions of spatial predicates (or for the temporal predicate in the next subsection) since they are self-explanatory. The directional relations include *left, right, above, below, front, back, south, north, west, east, northwest, northeast, southwest, southwest*, as well as the combinations of *front* and *back* with other directional relations. For example we can have *front_left, front_northwest*, etc. Precise definitions of directional relations can be found in [LÖS96]. The topological predicates include *inside, covers, touch, overlap, disjoint, equal, coveredBy*, and *contains* which are specified in [EF91] as eight fundamental topological relations. However, two pairs of predicates are inverses: *cover* vs *coveredBy* and *inside* vs *contains*. Figure 1 shows the basic six topological relations. Queries 5 and 6 illustrate how the spatial predicates can be used.

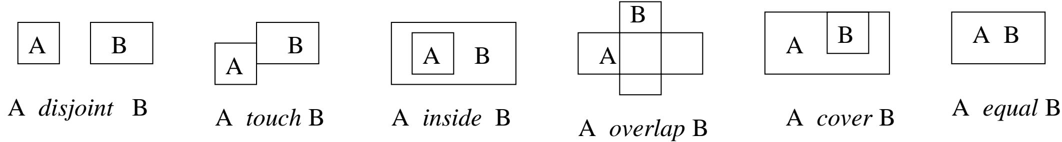


Figure 1: Definitions of Topological Relations

Query 9 Select all the cities, from a map of Canada, which are within 500km range of the longitude 60 and latitude 105 with populations in excess of 50000:

select *c*
from Maps *m*, *m*.cities *c*
where *m*.name="Canada" **and** *c*.location **inside circle**(**point**(60,105), 500) **and** *c*.population>50000

For each map, the method *cities* retrieves all the cities in this map. Then, a city's location is checked to see if it is within the required range. **point** is a constructor which accepts two values or three values to create a 2D point or 3D point respectively. Here, **point**(60, 105) represents a 2D point. **circle** is a circle object constructor which accepts a spatial point acting as the center of the circle and a radius. **inside** is a spatial predicate defined in Table 1. In MTQL, the same query can be specified as

```
select  c
from    m in C_map, c in m.B_cities
where   m.B_name.B_equal("Canada") and c.B_location inside
        circle(point(60,105), 500) and
        c.B_population.B_greaterthan(50000)
```

Query 10 Find all the names of the objects within a given region a in all images.

MOQL:

```
select  o.name
from    SalientObjects o, Images m
where   m contains o and (o inside a or a cover o)
```

MTQL:

```
select  o.B_name
from    o in C_salientObject, m in C_image
where   m contains o and (o inside a or a cover o)
```

The spatial predicates **inside** and **cover** both express the meaning of *within* (see Figure 1). The only difference is that **cover** shares boundaries of its arguments.

4.2 Spatial Functions

A spatial function computes attributes of an element or a set of elements of spatial objects. The spatial functions are shown in Table 2. The return type refers to the type of the object returned by a spatial function. Column *value* specifies the scale value. Function *mbr* stands for minimum bounding rectangle. In addition to these, there is a universal function *distance*, which returns a scale value when applied to any two spatial objects. Function *region* for both point and line objects allows a point or line to be converted to a region. Hence, all the predicates and functions for regions are applicable to points and lines. For example, directly checking a directional relation between a line and a region is not allowed. However, after the conversion of a line to a region, such a check can be made. Similarly we specify a set of pictorial functions:

- *pan*: view different portions of an image
- *resize*: change the size of an image
- *superimpose*: synthesize two images into one

Query 11 illustrates the use of the spatial functions while the use of the pictorial functions is illustrated in Query 15.

Return Type	point	line	region	value
point	nearest, farthest		region	
line	intersect	intersect	region	length, slope
region	centroid		interior, exterior, mbr	area, perimeter

Table 2: Spatial Functions

Query 11 Find the forests and their areas from the maritime regions where the forests are covered by provincial boundaries.
MOQL:

```
select  forest, area(forest)
from    Forests forest
where   forest.region coveredBy
          select  p.region
          from    Provinces p
          where   p.region coveredBy maritimeRegion
```

MTQL:

```
select  forest, area(forest)
from    forest in C_forest
where   forest.B_region coveredBy
          select  p.B_region
          from    p in C_province
          where   p.B_region coveredBy maritimeRegion
```

The above query illustrates the binding of two nested mappings combined with the spatial function **area** and spatial predicate **coveredBy**. The provincial region is passed from the interior level and used to direct the search in the exterior, one to produce those forests in the maritime provinces which are completely covered by provinces.

5 Temporal Primitives

The inclusion of temporal data in a multimedia query language is an essential requirement. Research in temporal queries has focused more on historical (discrete) databases rather than on databases of temporal media (e.g., [Sno95]). Thus, the focus has been on the reflections of changes of the representation of real world objects in

a database (e.g., President Clinton gave a speech at 2:00pm on July 4, 1996), rather than changes in continuous and dynamic media action. A complicated temporal SQL (TSQL2) [Se94] has been proposed as a possible standard for historical databases. Our interest is in temporal relationships among salient objects in multimedia data, not the real world historical relationships which are the major concern of TSQL2. A typical temporal multimedia query is "*Find the last clip in which person A appears*". The specification of the temporal relationship *last* needs special support from query languages to process this query.

A *time interval* is identified as the basic anchored specification of time. Allen [All83] introduces a set of 13 temporal interval relations which have been widely accepted. The 13 relations are *equal, before, after, meet, metBy, overlap, overlapedBy, during, include, start, startedBy, finish, finishedBy.*

5.1 Temporal Functions

Our choice of functional abstractions for temporal objects is influenced by the work of [GLÖS96]. Interval unary functions which return the lower bound, upper bound and length of the time interval are defined, while binary functions contain set-theoretic operations viz *union*, *intersection* and *difference*[2]. A time interval can be expanded or shrunk by a specified time duration.

A *time instant* is a specific anchored moment in time. A time instant is modeled as a special case of a (closed) time interval which has the same lower and upper bound, e.g., $Jan\ 24, 1996 = [Jan\ 24, 1996,$
$Jan\ 24, 1996]$. A wide range of operations can be performed on time instants. A time instant can be compared with another time instant with the transitive comparison operators $<$ and $>$. A time span can be *added* to or *subtracted* from a time instant to return another time instant. A time instant can be compared with a time interval to check if it falls before, within or after the time interval.

A *time span* is an unanchored relative duration of time. A time span is basically an atomic cardinal quantity, independent of any time instant or time interval. A time span can be compared with another time span using the transitive comparison

[2]Note that the union of two disjoint intervals is not an interval. Similarly, for the difference operation, if the second interval is contained in the first, the result is not an interval.

operators < and >. A time span can be subtracted from or added to another time span to return a third time span. We consider the following temporal granularity: *year, month, day, hour, minute, second, ms* (millisecond).

5.2 Continuous Media Functions

For continuous media, we consider only video data while audio will be considered in the future. We model a *video* as a sequence of clips and a *clip* as a sequence of frames. A *frame*, the smallest unit of a video object, can be treated as an image. Each frame is associated with a timestamp or time instant while a clip or a video is associated with a time interval. This implies that frames, clips, and videos can be ordered. Therefore, we can ask for the previous frame to a given frame or the last frame of a clip or a video. The continuous media functions are shown in Table 3. A universal function *timeStamp* applies to frames, clips, and videos and returns a time instant.

Return Type	frame	clip	video
frame	prior, next	clip	
clip	firstFrame, lastFrame, nth	prior, next	video
video		firstClip, lastClip, nth	

Table 3: Continuous Media Functions

Since video data consists of sequences of images, they share all the attributes of image data such as color, shape, objects, and texture. Unlike images, videos have temporal relations. Such temporal relations introduce dynamicity, (e.g. *motion*) which does not exist in image data. The implied motion in video data can be attributed to a camera (global) motion and an object (local) motion [ABL95]. In MOQL, an object motion is modeled by the multimedia database and then queried by using temporal predicates or functions. The definition of the abstract camera actions are based on [HK95]. A camera has six degrees of freedom representing translation along each axis (x: track, y: boom, z: dolly) and rotation about each axis (x: tilt, y: pan, z: rotate). In addition, a change in the camera's focal length produces scaling or magnification

of the image plane (zoom in and zoom out). To extract these features, each video stream should be first segmented into different logical units by locating *cuts* (camera breaks). Cuts can be classified into different categories, such as fade, wipe, dissolve, etc. We define the following camera motion boolean functions: *zoomIn, zoomOut, panLeft, panRight, tiltUp, tiltDown, cut, fade, wipe*, and *dissolve*.

In this section we assume that each continuous media object has a time interval associated with it that can be accessed through a method, *timestamp*. Furthermore, we assume that each salient object has a set of *timestamped physical representations*. The timestamped physical representation of a salient object indicates the physical characteristics of the salient object at different times. Typical physical characteristics of a salient object include geometric region, color, region approximation, etc. The set of physical representations of a salient object is accessible through method *prSet*. The following queries illustrate the temporal features of MOQL and MTQL.

Query 12 Find the last clip in which person *p* appears in the video *myVideo*:

```
select   lastClip(d)
from     d in (select c from myVideo.clips c
               where c contains p
               order by c.timestamp)
```

or

```
select   c
from     myVideo.clips c
where    c contains p and and (upperBound(c.timestamp) >=  all
                   select   upperBound(d.timestamp)
                   from     myVideo.clips d
                   where    d contains p)
```

The first solution uses the features of the video function **lastClip** and the OQL's **order by** clause. It is simpler than the second one. We assume that each video object has a method *clips* which returns a sequence of clips and each clip has a method *timestamp* which returns the time interval associated with this clip. Since two clips' intervals may overlap, we cannot simply rely on temporal predicates **after** or **meet** to perform the query. However, if we are sure that the upper bound of an interval is greater than or equal to all others, then its associated clip must be the last clip in a video. Therefore, we used a nested MOQL statement to express Query 12. In MTQL, the query can be expressed as

```
select   c
from     c in myVideo.B_clips
where    c contains p and and (upperBound(c.B_timestamp) >= all
                   select   upperBound(d.B_timestamp)
                   from     myVideo.B_clips d
                   where    d contains p)
```

Since TQL does not currently support the **order by** clause, we can only use the second solution for this query.

Query 13 List all the clip-pairs in which object p simultaneously appears.

MOQL:

```
select   c1, c2
from     Clips c1, Clips c2, p.prSet pr
where    c1 contains p and c2 contains p and
         pr.timestamp during intersection(c1.timestamp, c2.timestamp)
```

MTQL:

```
select   c1, c2
from     c1 in C_clip, c2 in B_clip, pr in p.prSet
where    c1 contains p and c2 contains p and
         pr.B_timestamp during
         intersection(c1.B_timestamp, c2.B_timestamp)
```

Note that because of video editing techniques such as fade and dissolve, clips can overlap in time. The tricky part of this query is in finding the overlap part of two neighboring clips. The temporal function **intersection** accomplishes this. Of course, object p must be within such an overlap and this constraint is achieved by using the temporal predicate **during**.

Query 14 List clips where person p_1 is at left of person p_2 and later the two exchange their positions:

```
select   c
from     Clips c, p1.prSet pr11, p1.prSet pr12, p2.prSet pr21, p2.prSet pr22
where    c contains p1 and c contains p2 and pr11 left pr21 and
         pr12 right pr22 and
         intersection(pr11.timestamp, pr21.timestamp) during
         pr11.timestamp and
         intersection(pr12.timestamp, pr22.timestamp) during
         pr12.timestamp and
         (pr11.timestamp before pr12.timestamp or
         pr11.timestamp meet pr12.timestamp)
```

Suppose clip c is the one we are looking for, then it contains both p_1 and p_2. In this case, both p_1 and p_2 must have at least two different physical representations

respectively: one is p_1 at left of p_2 and another one is p_1 at right of p_2. We use pr_{11} and pr_{12} to represent the two states of p_1, and pr_{21} and pr_{22} to represent the two states of p_2. The spatial constraints bind p_1 to the left of p_2 and p_1 to the right of p_2 while the temporal constraints bind the intersection of pr_{11} and pr_{21} (as well as pr_{12} and pr_{22}) to be not empty. Such temporal constraints guarantee that p_1 and p_2 appear together sometime in this clip. Certainly, the timestamp of the relation p_{11} at left of pr_{21} must be previous to the timestamp of the relation pr_{21} at right of pr_{22}. In MTQL

select c
from c **in** *C_clip*, pr_{11} **in** p_1.prSet, pr_{12} **in** p_1.prSet, pr_{21} **in** p_2.prSet, pr_{22} **in** p_2.prSet
where c **contains** p_1 **and** c **contains** p_2 **and** pr_{11} **left** pr_{21} **and**
pr_{12} **right** pr_{22} **and**
intersection(pr_{11}.*B_timestamp*, pr_{21}.*B_timestamp*) **during**
pr_{11}.*B_timestamp* **and**
intersection(pr_{12}.*B_timestamp*, pr_{22}.*B_timestamp*) **during**
pr_{12}.*B_timestamp* **and**
(pr_{11}.*B_timestamp* **before** pr_{12}.*B_timestamp* **or**
pr_{11}.*B_timestamp* **meet** pr_{12}.*B_timestamp*)

5.3 Presentation Functions

The query language has to deal with the integration of all retrieved objects of different media types in a synchronized way. For example, consider displaying a sequence of video frames in which someone is speaking, and playing a sequence of speech samples in a news-on-demand video system. The final presentation makes sense only if the speaking person's lip movement is synchronized with the starting time and the playing speed of audio data. Because of the importance of delivering the output of query results, supporting query presentation has become one of the most important functions in a multimedia query system. Both spatial and temporal information must be used to present query results for multimedia data. The spatial information will tell a query system what the layout of the presentation is on the physical output devices, and the temporal information will tell a query system the sequence of the presentation along a time line (either absolute time or relative time). We add a **present** clause as a direct extension to OQL as:

select [**distinct**] projection_attributes
from query [[**as**] identifier] {, query [[**as**] identifier] }
[**where** query]

[**present** layout { **and** layout }]

and to TQL as:

select object_variable_list
[**into** [**persistent** [**all**]] collection_reference
from range_variableList
where boolean_formula
[**present** layout { **and** layout }]

where, *layout* consists of three components:

- spatial layout which specifies the spatial relationships of the presentation, such as how many displaying windows, sizes and locations of the windows, etc.
- temporal layout which specifies the temporal relationships of the presentation, such as which media objects should start first, how long the presentation should last, etc.
- scenario layout which allows a user to specify both spatial and temporal layout using other presentation models or languages.

In order to support the spatio-temporal requirements of query presentations, we define the following presentation functions:

atWindow(identifier, point, point): setting a spatial layout within a window defined by two points;

identifier: an identifier of any media object;

point: a spatial point object, e.g. (10, 20).

atTime(absoluteTime): setting a real world time for supporting synchronization;

absoluteTime: e.g. 1996/11/8/13:40:00

display(identifier, start_offset, duration): present a non-continuous media object;

identifier: a graphical, image, or text object;

start_offset: time the display starts;

duration: display duration (default **forever**);

play(identifier, start_offset, duration, speed): present a continuous media object;

identifier: a frame, clip, video, or audio object;

start_offset: time the display starts;

duration: display duration (default: length of the identifier);

speed: playing speed factor (1.0: normal, 0.5: half normal speed, 2.0: double speed);

thumbnail(identifier): present the results as thumbnail objects;

identifier: any allowable media type;

resize(identifier, width, height): resetting the size of a media object;

identifier: an identifier of a media object;

width: the width of the result media object;

height: the height of the result media object.

parStart: two presentation events start simultaneously.

parEnd: as soon as one event ends, another event ends too.

after: an event starts right after another event finishes.

Note that most presentation functions are applied only to the first element of the result collection. Our experience tells us that no sophisticated query presentation is possible in OQL-based (or SQL-based) query languages. One major reason is that the return result is a collection of objects and a user cannot select a particular object since there is no handle to reference it. Another major reason is that OQL-based query languages cannot support interactive multimedia presentations. However, providing some basic functions for query presentation is very useful because in many cases, sophisticated presentation is not necessary. A typical example is searching for information using the World Wide Web on the Internet. For more sophisticated cases, presentation and synchronization requirements need to be specified outside of the query specification. However, MOQL and MTQL provide a mechanism to allow the invocation of such types of specifications. The following query examples show some features of the presentation functions.

Query 15 Find images, in which p appears, and display the result for 10 seconds with size (100, 100).

MOQL:

select m
from Images m
where m **contains** p
present **resize**(m, 100, 100) **and display**(m, 0, 10)

MTQL:

```
select   m
from     m in C_image
where    m contains p
present  resize(m, 100, 100) and display(m, 0, 10)
```

Function **resize** will change the size of the image to 100 pixels by 100 pixels. The offset of the **display** event is zero which means it will start immediately when the result is delivered. We may change the **present** clause into **thumbnail**(m) which will display thumbnail images as references to the real images.

Query 16 Find all the images and videos pairs such that the video contains all the cars in the image. Show the image in a window at ((0, 0), (300, 400)) and the video in a window at ((301, 401), (500, 700)). Start the video 10 seconds after displaying the image and display the images for 20 seconds, but play the video for 30 minutes.
MOQL:

```
select   m, v
from     Images m, Videos v
where    for all c in (select r from Cars r where m contains r)
         v contains c
present  atWindow(m, (0, 0), (300, 400)) and
         atWindow(v, (301, 401), (500, 700)) and
         play(v, 10, normal, 30*60) parStart display(m, 0, 20)
```

MTQL:

```
select   m, v
from     m in C_iamge, v in C_video
where    forAll c in (select r from r in C_car where m contains r)
         v contains c
present  atWindow(m, (0, 0), (300, 400)) and
         atWindow(v, (301, 401), (500, 700)) and
         play(v, 10, normal, 30*60) parStart display(m, 0, 20)
```

Operator **parStart** starts both video and image media objects simultaneously. Therefore, the image object will be displayed immediately. However, since the start offset time for the video is 10 seconds, the video object will start 10 seconds after the image object starts. We use a default value (**normal**) for video playing. This can be changed for faster or slower playing by choosing a number either bigger than one or less than one respectively.

Query 17 Find all clips in which both zoom-in and zoom-out exist and show their first frames in a thumbnail format.

MOQL:

```
select   firstFrame(c)
from     Clips c
where    zoomIn(c) and zoomOut(c)
present  thumbnail(firstFrame(c))
```

MTQL:

```
select   firstFrame(c)
from     c in C_clip
where    zoomIn(c) and zoomOut(c)
present  thumbnail(firstFrame(c))
```

Video special effects can be queried through the predicate **contains**. This results in a set of thumbnail images which are generated from the first frame of each clip.

6 Implementation of The Languages

TQL has been implemented as an extension of the first version of the TIGUKAT object model [ÖPS+95]. There is a current effort to implement a revised version of the TIGUKAT object model and TQL will be reimplemented within that extent. We have, therefore, deferred implementing multimedia extensions into TQL and decided to focus on OQL. We briefly discuss the current MOQL implementation. Figure 2 shows the overall architecture of MOQL query processing. Finished work is represented by the shaded boxes. MOQL queries are first examined syntactically by an MOQL parser which is written in LEX/YACC [LMB92]. A semantic checking procedure is conducted following the syntactic checking. The major work of this procedure is to validate types, classes, and class extents which are used in the MOQL query. It uses the underlying database system catalogs (or metadata) to enforce the correctness of data members, method invocations, or path expressions. Some query rewriting techniques should be performed to do query optimization at the MOQL level, such as renaming variables or eliminating nested queries [Kim82]. Then, an MOQL query is transformed into an object query algebra and an initial query tree, called *canonical* query tree, is generated. Many object query optimization techniques [ÖB95] can be applied to this canonical tree. Figure 2 (b) shows such a canonical query tree generated from **Query 6**. The algebraic operators (PROJECT, SELECT, and CROSS_PRODUCT) in Figure 2 (b) are standard relational algebraic operators; a multimedia object algebra is used in this step. The goal of query optimization is

find a query evaluation plan that minimizes the most relevant performance measure. After query optimization, an "optimized" query evaluation plan is produced. Using a simple tree traversal algorithm, this plan is compiled into a representation (executable code) ready for execution by the database system. Query optimization is the subject of our future research.

7 Conclusion

A powerful query language significantly helps users to manipulate a multimedia DBMS. It also helps to maintain the desired independence between the database and the application. Such a powerful query language should be as general as possible. However, most existing multimedia database systems are designed for specific applications. Therefore, the query languages are inherently restricted to a particular domain. This is not acceptable in a dynamically changing research area as new techniques emerge. The complex spatial and temporal relationships inherited in the wide range of multimedia data types make a multimedia query language quite different from its counterparts in traditional database systems. Surprisingly, spatial queries and temporal queries are not supported by most multimedia query systems. Furthermore, query presentation is much more complex than in traditional databases due to the synchronization of different media.

Our approach to designing a general-purpose multimedia query language is to extend the current standard query language OQL, which has been widely accepted by the ODBMS research and industrial communities, as well as research object query language TQL. The extended languages MOQL and MTQL can then be easily incorporated into existing ODBMSs. Users who are already familiar with OQL or TQL do not have to learn a new language. The extensions are accomplished by including spatial properties, temporal properties, and presentation properties. In particular, content-based spatial and temporal queries and query presentation are supported. An MOQL interpreter which parses MOQL queries and generates an algebraic tree is implemented.

There are two activities that we are currently carrying out. One is the development of a visual query interface built on top of MOQL. Even though MOQL provides powerful predicates, some multimedia queries are easier to specify visually. In an

ideal environment, MOQL will establish the basis of a visual query interface and serve as the embedded query language for application development. The second activity is to implement MOQL on top of an existing ODBMS. We are using ObjectStore [LLOW91] for this purpose. Further work needs to done in investigating the support for audio media, which is not addressed yet in MOQL. Once we identify the important primitives for audio data, the extension is straightforward. Another issue we will study is the optimization of executing MOQL queries.

Our future work on multimedia extensions to query languages will focus on structured multimedia documents which includes some international standards, such as SGML (Standard General Mark-up Language) and HyTime (Hypermedia/Time-Based Structural Language). Structured documents are important in any multimedia information systems, particularly for any system allowing the Internet access. One of interesting research problems in this area is how to enable users to query multimedia documents without complete document structural knowledge.

Multimedia DBMSs should use regular file systems (for efficiently handling traditional data) and multimedia servers (for efficiently handling multimedia data) as underlying storage systems and provide additional functions [CC95]. A high level query language is one such additional function and its support is essential in powerful multimedia DBMSs. We view the MOQL and the MTQL are a very important step in this direction.

Acknowledgment

We thank Dr. Vincent Oria for useful discussion in the process of preparing this report.

References

[AB91] R. Ahad and A. Basu. ESQL: A query lanugage for the relational model supporting image domains. In *Proceedings of the 7th International Conference on Data Engineering*, pages 550—559, Kobe, Japan, 1991.

[ABL95] G. Ahanger, D. Benson, and T. D. C. Little. Video query formulation. In *Proceedings of Storage and Retrieval for Images and Video Databases II,*

IS&T/SPIE Symposium on Electronic Imaging Science and Technology, pages 280—291, San Jose, CA, February 1995.

[All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of ACM*, 26(11):832—843, 1983.

[ATS96] H. Arisawa, T. Tomii, and K. Salev. Design of multimedia database and a query language for video image data. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 462—467, Hiroshima, Japan, June 1996.

[BRG91] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1):1—41, January 1991.

[Cat94] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Francisco, CA, 1994.

[CC95] S. T. Campbell and S. M. Chung. The role of database systems in the management of multimedia information. In *Proceedings of International Workshop on Multimedia Database Management Systems*, pages 31—39, Blue Mountain Lake, New York, August 1995.

[CIT+93] A. F. Cardenas, I. T. Ieong, R. K. Taira, R. Barker, and C. M. Breant. The knowledge-based object-oriented PICQUERY+ language. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):644—657, August 1993.

[CIT94] W. W. Chu, I. T. Ieong, and R. K. Taira. A semantic modeling approach for image retrieval by content. *The VLDB Journal*, 3:445—477, 1994.

[DG92] N. Dimitrova and F. Golshani. EVA: A query language for multimedia information systems. In *Proceedings of Multimedia Information Systems*, Tempe, Arizona, February 1992.

[EF91] M. Egenhofer and R. Franzosa. Point-set topological spatial relations. *International Journal of Geographical Information Systems*, 5(2):161—174, 1991.

[Ege94] M. Egenhofer. Spatial SQL: A query and presentation language. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):86—95, January 1994.

[GLÖS96] I. A. Goralwalla, Y. Leontiev, M. T. Özsu, and D. Szafron. Modeling time: Back to basics. Technical Report TR-96-03, Department of Computing Science, University of Alberta, February 1996.

[GR96] E. J. Guglielmo and N. C. Rowe. Natural-language retrieval of images based on descriptive captions. *ACM Transactions on Information Systems*, 14(3):237—267, July 1996.

[Güt94] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 297—308, Santiago, Chile, 1994.

[HK95] N. Hirzalla and A. Karmouch. A multimedia query specification language. In *Proceedings of International Workshop on Multimedia Database Management Systems*, pages 73—81, Blue Mountain Lake, New York, August 1995.

[KC96] T. C. T. Kuo and A. L. P. Chen. A content-based query language for video databases. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 456—461, Hiroshima, Japan, June 1996.

[Kim82] W. Kim. On optimizing an SQL-like nested query. *ACM Transaction on Database Systems*, 7(3):443—469, September 1982.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of ACM*, 34(10):19—20, 1991.

[LMB92] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[LÖS96] J. Z. Li, M. T. Özsu, and D. Szafron. Modeling of video spatial relationships in an object database management system. In *Proceedings of the International Workshop on Multimedia Database Management Systems*, pages 124—133, Blue Mountain Lake, NY, August 1996.

[MHM96] Elina Megalou, Thanasis Hadzilacos, and Nikos Mamoulis. Conceptual title abstractions: Modeling and querying very large interactive multimedia repositories. In *Proceedings of the International Conference on Multimedia Modeling*, pages 323—338, Toulouse, France, November 1996.

[MS96] S. Marcus and V. S. Subrahmanian. Foundations of multimedia database systems. *Journal of ACM*, 43(3):474—523, 1996.

[ÖB95] M.T. Özsu and J. Blakeley. Query optimization and processing in object-oriented database systems. In W. Kim, editor, *Modern Database Systems*, pages 146—174. Addison-Wesley, 1995.

[OM88] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14(5):611—629, May 1988.

[ÖPS+95] M. T. Özsu, R. J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. TIGUKAT: A uniform behavioral objectbase management system. *The VLDB Journal*, 4:100—147, 1995.

[OT93] E. Oomoto and K. Tanaka. OVID: Design and implementation of a video-object database system. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):629—643, August 1993.

[PLÖS93] R. J. Peters, A. Lipka, M. T. Özsu, and D. Szafron. An extensible query model and its languages for a uniform behavioral object management system. In *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 403—412, Washington D.C., USA, November 1993.

[PS95] P. Pazandak and J. Srivastawa. The language components of DAMSEL: An embedable event-driven declarative multimedia specification language. In *Proceedings of SPIE Photonics*, pages 121—128, October 1995.

[RFS88] N. Roussopoulos, C. Faloutsos, and T. Sellis. An efficient pictorial database system for PSQL. *IEEE Transactions on Software Engineering*, 14(5):639—650, May 1988.

[Se94] R. T. Snodgrass and et al. TSQL2 language specification. *SIGMOD Record*, 23(1):65—86, March 1994.

[Sno95] R. T. Snodgrass. Temporal object-oriented databases: A critical comparison. In W. Kim, editor, *Modern Database Systems*, pages 386—408. Addison-Wesley, 1995.

A Appendix: MOQL Specification

We follow the OQL grammar convention with only change: “⌊” and “⌉” are used to as an optional symbol to avoid confusion with symbols “[” and “]”.

A.1 Axiom

query_program ::= {define_query;} query

define_query ::= **define** identifier **as** query

A.2 Basic

query ::= **nil** | **true** | **false** | integer_literal | float_literal | character_literal

| string_literal | entry_name | query_name | bind_argument

| from_variable_name | (query)

A.3 Simple Expression

query ::= query + query | query **-** query | query * query

| query / query | **-** query | query **mod** query

| **abs** (query) | query || query

A.4 Comparison

query ::= query comparison_operator query | query **like** string_literal

comparison_operator ::= = | **!** = | > | < | >= | <=

A.5 Boolean Expression

query ::= **not** query | query **and** query | query **or** query | spatial_expression

| temporal_expression | boolean_expression

A.6 Constructor

query ::= type_name (⌊ query ⌉)

| type_name (identifier : query {, identifier :} query })

| **struct** (identifier : query {, identifier : query })

| **set** (⌊ query {, query }⌉)

| **bag** (⌊ query {, query }⌉) | **list** (⌊ query {, query }⌉)

| (query , query {, query }) | ⌊ **list** ⌉(query .. query)

| **array** (⌊ query , query }⌉)

A.7 Accessor

query ::= query * query
| dot attribute_name | query dot relationship_name
| query dot operation_name (query {, query }) | query [query : query]
| query [query] | **first** (query) | **last** (query)
| function_name ([query {, query }])
dot ::= **.** | − >

A.8 Collection Expression

query ::= **for all** identifier **in** query : query
| **exists** identifier **in** query : query
| **exists** (query) | **unique** (query)
| query comparison_operator quantifier query
| query **in** query | **count** (query) | **count** (*)
| **sum** (query) | **min** (query)
| **max** (query) | **avg** (query)
quantifier ::= **some** | **any** | **all**

A.9 Select Expression

query ::= **select** [**distinct**] projection_attributes
from variable_declaration {, variable_declaration }
[**where** query]
[**group by** partition_attributes]
[**having** query]
[**order by** sort_criterion {, sort_criterion }]
[**present** layout { **and** layout }]
projection_attributes ::= projection {, projection } | *
projection ::= query | identifier : query | query **as** identifier
variable_declaration ::= query [[**as**] identifier]
partition_attributes ::= projection {, projection }
sort_criterion ::= query [ordering]
ordering ::= **asc** | **desc**

A.10 Set Expression

query ::= query **intersect** query | query **union** query
| query **except** query

A.11 Conversion

query ::= **listtoset** (query) | **element** (query) | **distinct** (query)
| **flatten** (query) | (class_name) query

A.12 Spatial Expression

spatial_expression ::= spatial_function_expression comparison_operator
spatial_function_expression
spatial_expression ::= spatial_object spatial_predicate spatial_object
spatial_function_expression ::= spatial_term
[arithmetic_operator spatial_function_expression]
spatial_term ::= float_literal | integer_literal | spatial_object
| spatial_function (query)
spatial_function ::= **nearest** | **farthest** | **length** | **slope** | **centroid**
| **area** | **mbr** | **distance** | **intersect**
| **interior** | **exterior** | **perimeter**
spatial_predicate ::= generic_spatial_predicate | topological_predicate
| directional_predicate | picture_predicate
generic_spatial_predicate ::= **nearest** | **farthest** | **intersect** | **inside** | **cross**
| **centroid** | **midpoint**
topological_predicate ::= **equal** | **touch** | **inside** | **contains** | **overlap**
| **disjoint** | **cover** | **coveredBy**
directional_predicate ::= positional_predicate | depth_predicate
| simple_directional_predicate
| complex_directional_predicate
positional_predicate ::= **left** | **right** | **above** | **below**
depth_predicate ::= **back** | **front**
simple_directional_predicate ::= **north** | **south** | **west** | **east**
| **northwest** | **northeast** | **southwest** | **southeast**
complex_predicate ::= depth_predicate _ positional_predicate
| depth_predicate _ simple_directional_predicate
picture_predicate ::= **equal** | **coincident** | **subpicture** | **similar**
spatial_object ::= query | **point** (integer_literal , integer_literal)
| **line** (point , point) | **window** (point , point)
| **circle** (point , radius)

A.13 Temporall Expression

temporal_expression ::= temporal_object temporal_predicate temporal_object
| temporal_function_expression comparion_operator
temporal_function_expression
temporal_function_expression ::= temporal_term [arithmetic_operator
temporal_function_xpression]
temporal_term ::= integer_literal | float_literal | temporal_object
| temporal_function (query)
temporal_function ::= **union** | **intersection** | **difference** | **add** | **subtract**
| **upperBound** | **lowBound** | **length**
| **prior** | **next** | **nth** | **firstClip** | **lastClip** | **firstFrame**
| **lastFrame**
temporal_predicate ::= **equal** | **before** | **after** | **meet** | **metBy** | **overlap**
| **overlapedBy** | **during** | **include** | **start** | **startedBy**
| **finish** | **finishedBy**
temporal_object ::= query | **instant** (integer_literal) | **span** (integer_literal)
| **interval** (integer_literal , integer_literal)

A.14 Boolean Expression

boolean_expression ::= contain_predicate | boolean_function
contain_predicate ::= media_object **contains** identifier
media_object ::= identifier
boolean_function ::= boolean_function_name (media_object)
boolean_function_name ::= **cut** | **fade** | **wipe** | **dissolve** | **zoomIn** | **zoomOut**
| **panLeft** | **panRight** | **tiltUp** | **tiltDown**

A.15 Presentation Layout

layout ::= spatial_layout | temporal_layout | scenario_layout
spatial_layout ::= **atWindow** (identifier , point , point)
| image_function (identifier)
temporal_layout ::= event present_operator event
event ::= display_event | play_event | thumbernail_event
| event **atTime** (absoute_time)
display_event ::= **display** (identifier ⌊, start_offset ⌊, duration ⌉⌉)
play_event ::= **play** (identifier ⌊, start_offset ⌊, duration ⌊, speed ⌉⌉⌉)
thumbnail_event ::= **thumbnail** (identifier)
start_offset ::= integer_literal ⌊ temporal_granularity ⌉

duration ::= integer_literal ⌊ temporal_granularity ⌉| **forever**
speed ::= float_literal | **normal**
present_operator ::= **parStart** | **parEnd** | **after**
temporal_granularity ::= **year** | **month** | **day** | **hour** | **minute**
| **second** | **ms**
arithmetic_operator ::= $+$ | $-$ | $*$ | $/$
absolute_time ::= year / month / day / hour **:** minute **:** second
scenario_layout ::= user_presentation_specification

B Appendix: MTQL Specification

B.1 Basic Statements

MTQL_statement ::= select_statement | union_statement | minus_statement | intersect_statement
select_statement ::= **select** object_variable_list ⌊ **into** ⌊ **persistent** ⌊ **all** ⌉⌉ collection_reference ⌉
from range_variableList
⌊ **where** boolean_formula ⌉
⌊ **present** layout ⌉
union_statement ::= collection_reference **union** collection_reference
minus_statement ::= collection_reference **minus** collection_reference
intersect_statement ::= collection_reference **intersect** collection_reference
collection_reference ::= term | MTQL_statement

B.2 MTQL Terms

object_variable_list ::= object_variable ⌊, object_variable_list ⌉
object_variable ::= ⌊(cast_type)⌉ term | indexVariable
term ::= variable_reference | constant_reference | path_expression
index_variable ::= IDENTIFIER ⌊ behavior_name_list ⌉
variable_reference ::= IDENTIFIER
constant_reference ::= #IDENTIFIER
behavior_name_list ::= behavior_name | behavior_name_list , behavior_name
behavior_name ::= IDENTIFIER
path_expression ::= term **.** function_expression
function_expression ::= behavior_name () | behavior_name (term_list)
term_list ::= term ⌊, termList ⌉
varialbe_list ::= variable | variable_list , variable
range_variable_list ::= range_variable ⌊, range_variable_list ⌉
range_variable ::= variable_list **in** collection_reference ⌊+⌉
boolean_formula ::= atom | **not** boolean_formula
| boolean_formula **and** boolean_formula
| boolean_formula **or** boolean_formula
| (boolean_formula) | exists_predicate
| for_all_predicate | spatial_expression | temporal_expression
| boolean_expression
atom ::= term **=** term | term_list **in** collection_reference ⌊+⌉
exists_predicate ::= **exists** collection_reference

for_all_predicate ::= **forAll** range_variable_list boolean_formula

B.3 Spatial Expression

spatial_expression ::= spatial_function_expression comparison_operator
spatial_function_expression
spatial_expression ::= spatial_object spatial_predicate spatial_object
spatial_function_expression ::= spatial_term
[arithmetic_operator spatial_function_expression]
spatial_term ::= float_literal | integer_literal | spatial_object
| spatial_function (query)
spatial_function ::= **nearest** | **farthest** | **length** | **slope** | **centroid**
| **area** | **mbr** | **distance** | **intersect**
| **interior** | **exterior** | **perimeter**
spatial_predicate ::= generic_spatial_predicate | topological_predicate
| directional_predicate | picture_predicate
generic_spatial_predicate ::= **nearest** | **farthest** | **intersect** | **inside** | **cross**
| **centroid** | **midpoint**
topological_predicate ::= **equal** | **touch** | **inside** | **contains** | **overlap**
| **disjoint** | **cover** | **coveredBy**
directional_predicate ::= positional_predicate | depth_predicate
| simple_directional_predicate
| complex_directional_predicate
positional_predicate ::= **left** | **right** | **above** | **below**
depth_predicate ::= **back** | **front**
simple_directional_predicate ::= **north** | **south** | **west** | **east**
| **northwest** | **northeast** | **southwest** | **southeast**
complex_predicate ::= depth_predicate _ positional_predicate
| depth_predicate _ simple_directional_predicate
picture_predicate ::= **equal** | **coincident** | **subpicture** | **similar**
spatial_object ::= query | **point** (integer_literal , integer_literal)
| **line** (point , point) | **window** (point , point)
| **circle** (point , radius)

B.4 Temporall Expression

temporal_expression ::= temporal_object temporal_predicate temporal_object
| temporal_function_expression comparion_operator
temporal_function_expression

temporal_function_expression ::= temporal_term [arithmetic_operator
temporal_function_xpression]
temporal_term ::= integer_literal | float_literal | temporal_object
| temporal_function (query)
temporal_function ::= **union** | **intersection** | **difference** | **add** | **subtract**
| **upperBound** | **lowBound** | **length**
| **prior** | **next** | **nth** | **firstClip** | **lastClip** | **firstFrame**
| **lastFrame**
temporal_predicate ::= **equal** | **before** | **after** | **meet** | **metBy** | **overlap**
| **overlapedBy** | **during** | **include** | **start** | **startedBy**
| **finish** | **finishedBy**
temporal_object ::= query | **instant** (integer_literal) | **span** (integer_literal)
| **interval** (integer_literal , integer_literal)

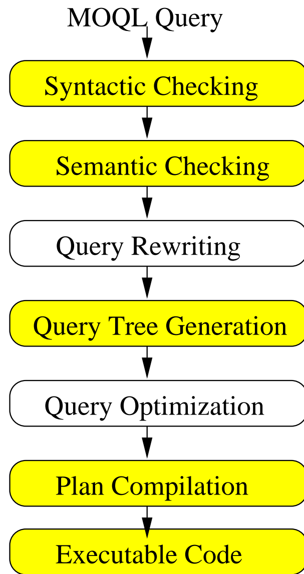
B.5 Boolean Expression

boolean_expression ::= contain_predicate | boolean_function
contain_predicate ::= media_object **contains** identifier
media_object ::= identifier
boolean_function ::= boolean_function_name (media_object)
boolean_function_name ::= **cut** | **fade** | **wipe** | **dissolve** | **zoomIn** | **zoomOut**
| **panLeft** | **panRight** | **tiltUp** | **tiltDown**

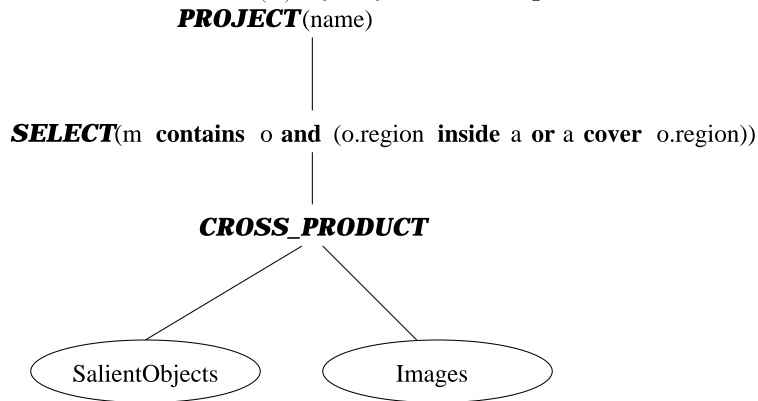
B.6 Presentation Layout

layout ::= spatial_layout | temporal_layout | scenario_layout
spatial_layout ::= **atWindow** (identifier , point , point)
| image_function (identifier)
temporal_layout ::= event present_operator event
event ::= display_event | play_event | thumbernail_event
| event **atTime** (absoute_time)
display_event ::= **display** (identifier ⌊, start_offset ⌊, duration ⌉⌉)
play_event ::= **play** (identifier ⌊, start_offset ⌊, duration ⌊, speed ⌉⌉⌉)
thumbnail_event ::= **thumbnail** (identifier)
start_offset ::= integer_literal ⌊ temporal_granularity ⌉
duration ::= integer_literal ⌊ temporal_granularity ⌉| **forever**
speed ::= float_literal | **normal**
present_operator ::= **parStart** | **parEnd** | **after**
temporal_granularity ::= **year** | **month** | **day** | **hour** | **minute**
| **second** | **ms**

arithmetic_operator ::= + | − | * | /
absolute_time ::= year / month / day / hour : minute : second
scenario_layout ::= user_presentation_specification



(a) Query Processing



(b) The Canonical Query Tree of **Query 10**

Figure 2: Query Processing and Query Tree