A Bi-Level Object-Oriented Data Model for GIS Applications

by

Amelia Yin Ling Choi

SIMON FRASER UNIVERSITY

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

July 1990

APPROVAL

Name: Amelia Yin Ling Choi

Degree: Master of Science

Title of Thesis: A Bi-Level Object-Oriented Data Model for GIS Applications

Examining Committee:

Dr. Ze-Nian Li, Chairman

Dr. Wo-Shun Luk
Senior Supervisor

Dr. Jia-Wei Han
Supervisor

Dr. Tom Poiker
External Examiner

July 3, 1990
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

A Bi-Level Object-Oriented Data Model for GIS Applications.

Author: _____
(signature)

Amelia Yin Ling Choi
(name)

July 12, 1990
(date)

ABSTRACT

In this thesis, a bi-level object-oriented data model, together with a user query language called OFQL (Object-oriented Functional Query Language) using function interface is designed which can well support applications like GIS (Geographic Information System). The data model is divided into two layers, the geographic object data model and the geometric object data model. The former primarily consists of the geographic objects, a set of semantic spatial functions, and some abstraction hierarchies through which the topological relationships of geographic objects are defined or derived. The geometric object data model has geometric objects which are the actual spatial representations of the geographic objects in the geographic object data model. It has also a set of geometric functions that retrieve and compute for geometric objects. With the OFQL user-interface, the user is able to pose spatial and non-spatial queries of a geographic nature without knowing the details at the geometric level. The general architecture of a GIS system using our data modeling approach consists of two modules, namely: the query processor and the function implementor.

Different issues in query processing, searching techniques and optimization for the function interface of OFQL in an object-oriented environment are discussed using QP graph as a tool for query processing plan generation. We will show how abstraction hierarchies in the geographic object data model, i.e. IS-A and PART-OF, can be elegantly handled without complicating the processing of a query. A new query processing scheme which allows both edge and chain processing is also introduced. Using the concept of cascaded function execution, both cyclic and acyclic chain processing can be handled in the same way. Finally, our QP graph can have more than one target object class which is of utmost importance for the generation of tuple objects reflecting new relationships among two or more geographic objects computed at run-time.

ACKNOWLEDGEMENTS

The myriad ideas which were once abstract have finally concurred into reality. I would like to dedicate this thesis to all persons whose sincere encouragement and thoughtful guidance have contributed greatly to the zenith of my achievement. Indeed, the very essence of their commendable support deserves my warmest gratitude and appreciation.

First of all, I would like to extend my whole-hearted gratitude to my senior supervisor, Professor Wo-Shun Luk. Without his thoughtful guidance throughout this research, his generous and constructive suggestions and comments, and his sincere words of encouragement, this thesis would not have been finished within a relatively short period of time.

Second, my sincere appreciation to Professor Tom K. Poiker and Professor Jia-Wei Han who have given me helpful comments and have supplied me with relevant reference materials.

Above all, my deepest gratitude to my beloved parents, brother, and sisters. The potpourri of my gratitude, appreciation, and feelings are more than words can say. Indeed, their love, care, concern, encouragement, and moral support embody the sole spirit and inspiration of this thesis.

Finally, I offer my entire success and glory to my Almighty God who is the shepherd and beacon of my daily path. Throughout my whole career, He has given me wisdom, faith, encouragement, guidance, and strength, without which the completion of this thesis is absolutely impossible.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1. Basic GIS Concepts

Geographic Information System (GIS) is an emerging technology and is a popular topic in Earth Science and other related application areas like forestry, natural resource management, municipal planning, wildlife habitation and many others. It is not a pure computer cartography or automated map-making system, nor is its capability limited to the graphical power of CAD. A conventional DBMS which basically selects and reports relevant information stored in a database is not adequate to handle this challenging application. The distinguished feature of GIS is in its capability of conducting *spatial searches* and *overlays*, and association of the spatial data with the non-spatial data to eventually generate *new* information.

Basically, we have two major types of objects for GIS applications. They are the geographic objects and the geometric objects. *Geographic objects* refer to the actual physical features or logical groupings of physical features on the earth's surface which are assigned a distinct name in real-life applications. Examples of these are countries, cities, zones, roads, rivers, soil, forests, bridges, drainage channels, recreational areas, and many others. Each of these entities could be described by a set of attributes like the name of a country, the driving speed of a street, the type of soil, some census data as in the population of a province, the average income within a certain district and others. These geographic objects are physically present on a specific area on the earth's surface, and thus, require a spatial representation to define exactly their locations.

Generally, there are two ways of representing the spatial data associated with these geographic objects, namely: vector representation and grid-cell or raster representation. Some of the basic types of entities in *vector representation* are the polygons (which may be used to represent forests or countries, for example), lines (which represent streets and others), and nodes (that represent wells, street intersections and others). People use vector format when doing analysis that requires high precision or in network analysis. We call these polygons, lines, and nodes as the *spatial objects* or *geometric objects* in this thesis. The *grid-cell* or *raster representation*, on the other hand, is another way of portraying spatial information with the use of

cellular organization. This representation is attractive in performing thematic overlays which will be described later on. In most of the existing GIS systems nowadays, conversion from vector representation to raster representation and vice versa is possible.

Some of the most commonly used *spatial searches* which are required for analysis are *containment*, *intersection*, and *proximity searches* like nearest, adjacent and others. Different spatial indexing mechanisms are designed to facilitate rapid access of relevant spatial data to perform these spatial searches efficiently. There is an active ongoing research on this particular area [Samet, H. 88]. The most popular spatial index structures are perhaps the R-tree [Guttman, A. 84] for spatial data stored in vector representation, and the quad-tree [Samet, H. 84] for those that are stored in raster format.

In the original way of doing geographic analysis, people have several map layers portraying the same physical area on earth, each of which represents a distinct theme. Examples of these themes are political boundaries, land use/land cover, census tract and enumeration areas, county and municipal boundaries, hydrography, transportation , soil type, vegetation and many others. *Spatial overlays* are often performed to see which physical area(s) satisfies all the conditions in the different themes. In the manual way of processing, these map layers or map sheets that represent the different themes of the same physical area are put on top of one another to perform the overlay. It is due to this method of organizing map sheets and themes that spatial overlays are sometimes called the *thematic overlays*. The cellular organization of the raster representation could do this type of overlay in the most natural and rapid way. Only those cells which have the specified code representing a particular characteristic or condition that is of interest to the user in the different themes are selected. This could be seen as a logical AND operation on each cell representing the same physical area in the different map layers. With the spatial searches and overlays, new information is generated and the non-spatial data should be able to relate to these spatial objects to provide an integrated view of a geographic object.

1.2. Objectives

In this thesis, we are not concerned with the optimization issues of spatial data representation, nor do we suggest the specific spatial indexing methods or algorithms that are to be used in performing spatial computation. Our objective is to develop a bi-level object-oriented data model that facilitates design and implementation of GIS applications. It consists of two layers, namely: the geographic object data model and the geometric object data model. The details of which will be given in the subsequent chapters of this thesis. We stress that our aim is not to provide a complete object-oriented data model wherein data definition

language (DDL), data manipulation language (DML), object identifier generation, and actual object manipulation including updates are to be handled. Instead, our emphasis is on how our modeling approach can provide abstraction to the high-level user such that user's view of geographic objects will be totally independent from their underlying spatial representation and computation. In fact, most of the modeling concepts and the bi-level data modeling approach are also applicable to other applications which consist of both the spatial data as well as the non-spatial data. In relation to that, we will design a high-level user query language using function interface called OFQL to allow user to pose spatial and non-spatial queries of a geographic nature. The general architecture of a GIS system using our bi-level data modeling approach will be designed. Spatial queries on geographic objects are automatically mapped by the query processor to some geometric functions applied on their corresponding spatial objects in the geometric object data model, with the function implementor deciding on the actual routines for executing each of these functions. Mapping between the two layers of the bi-level data model and implementation details of the query processor and the function implementor will not be discussed. However, assuming the use of a single sequential processor, an attempt is made to explore some query processing and optimization issues of OFQL.

1.3. Motivation and Related Work

There are many commercially available GIS systems, most of which were originally designed for mapping applications. Many of them have since evolved to provide sophisticated geographic analysis, and ARC/INFO [ESRI], a commercial product of ESRI (Environmental Systems Research Institute) for GIS applications, is perhaps one of the most well-known systems.

Its coverages are basically organized in two schemes, namely: *by location* into map sheets or map tiles, and *by content* into layers or themes. We could think of a world map as a jig-saw puzzle piecing together the different map sheets representing different countries. Layer, on the other hand, may refer to the theme of an entire map sheet.

ARC/INFO stores spatial data in external files and does not store them together with the non-spatial data in a database. As the name implies, the ARC file contains all the spatial information, which includes topological relationships and actual spatial representation like coordinates. The INFO component contains only the topological relationships and other non-spatial attributes, but not the actual spatial representation in the database. Topological relationship explicitly specifies the left and right polygons, starting and ending nodes of a given edge. Its primary application is in network analysis.

From the data modeling point of view, ARC/INFO is map-sheet-oriented, rather than geographic-entity-oriented. Using map tile as a unit has a direct correspondence to the manual process of doing geographic analysis. The discrete map-tile boundary complicates the computation and generation of topological relationships of geographic entities which lie at the boundary or run across map sheets. The system does not provide an integrated view of the geographic entities in its underlying structure which may also be a cause of inefficiency if the user wants to compute new information or to retrieve all information about a particular geographic entity. Indeed, there is no easy way for the user to pose ad hoc queries since there is no query language facility.

A more recent approach is to extend a conventional DBMS to provide the capability of handling spatial data at the query level [Ooi, Sacks-Davis and McDonell 89]. Though similar to ARC/INFO, it has a separate storage of spatial and non-spatial data, but its spatial data are not divided into map sheets. Each spatial object is treated as a unit of reference. It is through the unique geographic identifier or *gid* of each spatial object that associates it with the non-spatial object.

It has a query language called GEOQL, which is an extended SQL that provides a set of spatial operators to allow user to interface with the system and to write spatial queries. Its spatial representation is not completely abstracted in the query.

With the advent of object-oriented technology, GIS researchers have begun to experiment with object-oriented data modeling. OSAM* [Su, Krishnamurthy and Lam 89], for example, is an object-oriented semantic association model used by Martin Feuchtwanger [Feuchtwanger 87] for GIS applications. It includes five types of semantic associations, namely: aggregation, generalization, interaction (similar to relationship in an ER model), composition and finally, crossproduct association among domain objects. In his example, geographic objects like region which is a generalization of different entity objects: roads, districts and blocks, all of which refer to a particular area on the earth's surface, is seen as an aggregation of polygons.

OSAM* has a query language called OQL which requires user to specify the entire path of entity classes traversed based on certain grouping scheme in the CONTEXT clause. For example: COUNTRY -> PROVINCE -> CITY -> STRIP -> ... User has to be aware also of the spatial representation since aggregation association is used in the model to describe the association between the geographic and the spatial objects. Thus, to perform spatial queries, the user has to explicitly specify the spatial object in the CONTEXT clause.

In our opinion, none of the models make a clear distinction between the geographic objects and their underlying spatial representation. Moreover, there is no explicit mechanism to model the spatial relationships of the *geographic* objects.

In this thesis, we propose a bi-level object-oriented data model which provides the separation between the geographic and the geometric layers. The primary motivation is to support high-level GIS applications (e.g. decision-making in municipal planning and others) where user has to interface only with the geographic object data model without the need of knowing the underlying spatial representation of the geographic objects, i.e. raster or vector representation. Independence between these two layers ensures that the semantic meaning of the geographic object data model can always remain intact regardless of the underlying spatial object representation. With this bi-level data model, together with our high-level query language OFQL, user can pose ad hoc queries of a geographic nature that are relevant to his applications. Furthermore, in our proposed architecture, the query processor can automatically do the essential mappings between the two layers of our bi-level data model, and the function implementor interacts with the underlying storage which could be implemented as a relational database, an object-oriented database or others.

Unlike ARC/INFO, our model is not map-sheet or map-layer-oriented but object-oriented, which allows easy manipulation of individual objects. Our bi-level approach, together with the OFQL interface provides user with total abstraction from the spatial representation which is different from [Ooi, Sacks-Davis and McDonell 89] approach. We choose a function interface for our OFQL query language because functions can model well computations and attributes of objects, and relationships or mappings among objects (which include the spatial relationships of geographic objects). In fact, unlike OSAM*, mapping between geographic and spatial objects in our data model is not treated as aggregation but is handled by the query processor. Spatial object is not PART-OF a geographic object, but is just a representation of the geographic object. Aside from the IS-A hierarchy which models the non-spatial aspects or roles of the geographic objects, we also include PART-OF abstraction in our model to model total containment association among geographic objects. In other words, the abstraction hierarchies are used independently in each layer of the bi-level data model and are not used to model relationships among objects between the two layers. Our query processor also automatically deciphers the abstraction hierarchies that have to be traversed and uses them to generate optimized query processing plans, as opposed to OSAM*'s approach where user has to enumerate all the entity classes along the path of these hierarchies.

Finally, a new query processing scheme is introduced for the function interface of OFQL in an object-oriented environment. That is, depending on the appropriate heuristic rules applicable, query processor can

decide whether processing a single function (edge) as a unit or processing by groups of functions (chains) is more efficient. The motivation for introducing chain processing is to allow possible application of existence check and depth-first search technique in processing a query which offers more flexibility (especially in an object-oriented environment where clustering and object caching are supported) than the use of nested-for loops. In this case, less number of disk accesses, and reduction in both memory space and computations may be achieved. Furthermore, as a direct consequence of our method used, that is, cascaded function execution, both cyclic and acyclic chain processing can be handled in the same way and can benefit from reduction of search space such that only relevant objects are being considered at the earliest stage possible during query execution.

1.4. Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses the new features provided by our bi-level object-oriented data model together with its three modeling constructs. Chapter 3 describes the overall architecture of a GIS system using our bi-level data modeling approach. Chapter 4 discusses the components in each layer of the bi-level data model. Chapter 5 gives a description of OFQL. Chapter 6 discusses query processing strategies and query optimization of OFQL. Chapter 7 suggests some future research directions related to the model proposed in this thesis, and finally, conclusion is given in Chapter 8.

Chapter 2
Object-Oriented Data Model with Function Interface

Object-oriented database is the new technology for the non-traditional database applications like CAD, CASE, hypermedia systems and other engineering applications. To name a few of the ongoing projects and proposed data models using an object-oriented approach, we have GemStone Data Management System [Bretl et al. 89], ORION [Kim, Bertino and Garza 89], [Banerjee et al. 87], IRIS [Fish et al. 87], [Wilkinson, Lyngbaek and Hasan 90], ENCORE [Smith and Zdonik 87], FUGUE [Heiler and Zdonik 88a], [Heiler and Zdonik 88b], and OZ+ [Weiser and Lochovsky 89].

The object-oriented data model described in this thesis is somewhat similar to IRIS. The IRIS data model is primarily based on three constructs, namely: objects, types and operations, which directly correspond to objects, object classes, and functions of our bi-level data model respectively. We extended IRIS data model by including a new query processing scheme which allows chain processing in carrying out complex computations aside from the basic selections, retrievals, manipulations and simple aggregations modeled via functions associated with the object classes. This is of special importance in performing spatial searches and overlays at run-time. A new query language called Object-oriented Functional Query Language (OFQL) is also designed to handle both spatial and non-spatial queries.

Why is the object-oriented data model suitable for GIS applications? Queries in GIS are usually object-oriented. For example, planning activities like urban planning, natural resource management and other similar activities of a country are usually done within a country. From the federal level, the government may then be interested in the statistics of the provinces, with the provincial or even municipal government working out the details of each city, town, or municipality. At each level, a particular entity or what we call as an *object* like *city*, is of primary interest at a certain point of time. The relationship of these objects at different levels based on a particular grouping scheme such as political or administrative boundaries forms a hierarchy of geographic objects, which is an analogy to the notion of *complex object* in object-oriented database terminology or the PART-OF relationship in semantic data models. Each object class is a group of objects of similar structure and has its own set of operations that could be used for retrieval and computation. Each object is distinct and has its own distinct object identifier. This allows us to handle different versions of

a geographic object as it changes its spatial or non-spatial properties over a certain time frame. Similar to other semantic data models, inheritance is supported for the IS-A or generalization abstraction. The object-oriented data model provides not only inheritance of attributes or properties but also operations from superclass to subclasses.

For the rest of this section, we first introduce the general features of our bi-level data model in Section 2.1. The modeling constructs that are common to both layers of our bi-level data model will also be described in Section 2.2.

2.1. Features and Assumptions

Before we proceed to the discussion proper, let us briefly examine the assumptions and features of our new data model for GIS applications. The details of which will be discussed in Chapter 4.

- There is no notion of map tile or map sheet. All map layers are integrated into one basic map.
- The spatial representation of each geographic object for doing GIS analysis is independent from its graphical representation. All the spatial representations use the same scale of measurement.
- The availability of facilities for raster-vector format conversion.
- Distinct *spatially-associated* geographic objects occupying the same physical space on earth's surface, will be represented by distinct geometric objects for their spatial representation. Each of them are assigned a unique object identifier.
- We have a *bi-level data model* which completely abstracts user from the underlying spatial or geometric representation and the computation of geographic objects.
- OFQL is a powerful query language that answers queries of a geographic nature. Its basic building blocks are functions.
- Superfunctions could be used together with the built-in set of functions of the data model in defining more complicated application-oriented functions for the user. This provides extensibility to the set of functions available in modeling different GIS application domains.
- New query processing scheme allows processing by chains and/or edges. Different searching techniques (depth-first search and breadth-first search) can be used in processing chains. Abstraction hierarchies in the geographic object data model are used in query processing.

2.2. Three Modeling Constructs

The data model is primarily based on the IRIS data model, with three modeling constructs, namely: *objects*, *object classes*, and *functions*. The discussion below holds for both geographic and geometric layers of our bi-level object-oriented data model.

2.2.1. Objects

Objects represent entities. For GIS applications, we may have geographic objects in our geographic object data model, geometric objects in our geometric object data model (which are the spatial representations of the geographic objects), and primitive objects (which are of system built-in types like INTEGER, REAL, STRING, RANGE and BOOLEAN). Each object has its own identity. They are used as input/output (I/O) arguments of functions.

Objects could be generally classified into two types, namely: primitive and non-primitive which correspond to the literal and non-literal objects of the IRIS model respectively. The non-primitive objects include all the geographic objects and geometric objects in our data model and could be referenced using property values or what we call *attributes*, and relationships with other objects, all of which are modeled with the use of functions. An instance of an object class is called an object. We will use object and instance interchangeably in this thesis.

2.2.2. Object Classes

Object classes organize objects by structure and behavior. Different object classes could be organized to form IS-A and PART-OF hierarchies. The functions defined in different object classes may have identical names even though their semantics and underlying implementations are different.

The declaration below of a sample non-primitive object class COUNTRY will be used to introduce our simple data definition language for the model. Additional elements included in our data definition language that specify the different abstraction hierarchies of our data model will be introduced as we discuss them in detail in the subsequent chapters.

create objectclass COUNTRY
Name: STRING [1]
Capital: CITY [1]
Population: Sum(Population of PROVINCE)
PoliticalAllies: COUNTRY

The underlined keywords *create objectclass* create a new object class COUNTRY. The left hand side of the colons are the function names, while on their right, specify the domains which the functions will map to. The different types of domain specifications include primitive object classes (which are the built-in data types), non-primitive object classes and functions. Properties or attributes like *Name* and *Population* in the example above are modeled by functions. Given the function *Name*, it will be mapped to a value of STRING data type that gives us the name of the country. A function could also model some relationships which map an object to another object(s) of the same or different object class(es). In our example, *Capital* and *PoliticalAllies* functions have CITY and COUNTRY object classes as their domains respectively. The [1] specifies that there should be one and only one object or value returned by the function for each of the instances in the object class. Note that by default, we always assume that results of functions are set-valued, and thus, may consist of 0 or more elements in a set.

The function, *Population*, on the other hand, has its domain defined in terms of another function, that is, Sum(Population of PROVINCE). It means that the population of a country is equal to the sum of the population of all its provinces. This introduces the selective upward aggregation feature of our PART-OF abstraction hierarchy wherein the attribute of an aggregate parent object is defined by performing an arithmetic computation on a specific attribute of all its component objects which are PART-OF related to it. We will have the detail discussed in Section 4.1.2.

A *relationship object class* is an object class which instances are called *tuple objects*. The components of each tuple object are instances from two or more non-primitive object classes where there exists a certain mapping or relationship among them. Tuple objects can be explicitly stored or precomputed at the initial creation of the database or the mapping among the components of each tuple object can also be computed by a function or a query. We call the object classes of each of the components in each tuple object as a *non-primitive component object class* of the relationship object class.

Geographic and geometric object classes that are defined at the initial creation of the database are persistently stored and will not be deleted from the database unless explicitly specified. *New* objects could also be generated as a result of processing a user query. These are called the *query-generated objects*. They are by default *transient* which means that they exist only for a certain period of time but will be lost afterwards, or they could be *persistent* if the user chooses to explicitly store them in the database. In our model, these query-generated objects exist throughout the lifetime of a user session if given explicit declaration, thus, they could be referenced by some other queries in the same session via the query-generated object class name. Here, query-generated objects may either belong to a previously defined object class or to

a query-generated object class. In the former case, the object class is predefined before hand and the query-generated objects are *added* to that object class if the user explicitly specifies that they are to be persistently stored.

Below are the three cases by which the instances of a query-generated object class can be generated:

- A *query-generated object class* may consist of instances from only one object class that satisfy all the constraints imposed on them in user query. This could be seen as a selection (in the sense of relational algebraic operation) on the objects that are previously defined and stored in the database. In this case, this query-generated object class will have pointers to all these selected objects.

- A query-generated object class may also consist of several component object classes. In this case, it is actually a relationship object class. For each instance or the so called tuple object of this query-generated object class, it consists of an object from each of these non-primitive component object classes that are connected by a certain relationship among them. The attributes and abstraction hierarchies and functions of the component objects are not applicable to the entire tuple object as a whole, but could only be applicable to its corresponding component objects. Further discussions and examples on query-generated object classes are found in Section 5.1.

- Finally, it is also possible that results produced are newly generated, i.e., they are not previously stored in the database. An example of this will be to find the exact patch of land where we could find stable soil in a given parcel. This means that we are interested in the overlapping area of these two geographic objects. In this case, this new patch of land or LAND object will belong to a distinct object class, say, LAND, which is created at run-time if it is not previously defined. This LAND object will have the function values of the two objects from which it is derived. It will thus possess the function, Soiltype = "loam" from the SOIL object and other functions of the given PARCEL object. Note that this newly generated LAND object could be just a portion of the parcel, so it will also have its new distinct associated spatial representation at the geometric object data model created. This newly generated geometric object is then the query-generated geometric object which belongs to some predefined geometric object class.

2.2.3. Functions

Functions model *methods* (which are operations or computations that can be done on objects of individual object classes), *stored or computed relationships among objects* (which are mappings among objects of the same or different object classes including the modeling of the abstraction hierarchies), and *attributes* of individual non-primitive objects or tuple objects. In other words, functions could be applied to objects to obtain their properties, to perform computations, to test constraints or conditions and to perform mappings to yield related objects of the same or different object classes. It is in these powerful modeling capabilities of functions that make them an attractive and appropriate modeling construct that can be used in an object-oriented data modeling environment. Thus, our OFQL user query language uses a function interface.

Functions could be *stored* or precomputed in the database as tuple objects, or they may be *derived* using simple or nested functions similar to the following examples:

- Population^{-1}(1000) -> CITY

  This is actually an inverse function derived from the function, *Population(CITY) -> INTEGER*. Note that the results of the derived functions could be directly obtained from the stored information in the database. This function returns all the CITY objects with a population of 1000.

- Contains^{-1}(Population^{-1}(1000) -> CITY) -> COUNTRY

  This nested function gets all the cities with a population of 1000 and returns the COUNTRY object which contains these cities as the final result.

Nested functions are described in Section 4.1.3 when we discuss superfunctions.

Functions could also be *computed* at run-time. Here, functions have to perform some simple or complex computations in addition to pure retrieval, which will eventually generate new information that is not previously stored in the database, and thus, creating new query-generated objects. Simple examples of these include spatial operations like finding the exact area where two geographic objects in the geographic object data model overlap or finding the intersections of edges in the geometric object data model.

The general format of functions is as follows:

fn(input arguments) -> output arguments

where *input arguments* could be constants, or object class names.

The specific format of a valid function is as follows:

- fn({|cons|/|OC|[,]}) -> OC *or*
- fn(OC) comparator cons

where:

- function name (fn): first letter is capitalized, with the others printed in either upper or lowercase provided not all of them are in capital letters
- constant (cons): it could be a numeric constant or a string constant that is enclosed with double quotes
- object class name (OC): all letters are in uppercase
- []: all entities within [] could be optionally specified
- {}: all entities within {} could be repeated
- /: any one of the entities delimited by double bars || that are separated by a slash / in between must be chosen
- comparator: <, >, <=, >=, =, <>

Below are examples using the two formats of a valid function:

1. Contains(FOREST) -> LAKE

2. Type(DISTRICT) = "residential"

The first example generates all the LAKE objects that are spatially contained in any objects belonging to the FOREST object class. Note that the argument specified after -> indicates the output object class to be mapped to by the function. If -> is not specified, then by default, it is interpreted as a comparison function, which returns a boolean result. A TRUE will be returned if the comparison is satisfied. The second example is an example of comparison function which returns a boolean result, TRUE, if the specified DISTRICT object is of "residential" type, otherwise, FALSE is returned.

Chapter 3
Architecture

In this section, we will describe the general architecture of a GIS system that uses our bi-level object-oriented data modeling approach. There are two basic modules above the physical database, namely: the query processor and the function implementor. Please refer to Figure 3-1.

The query processor maps the user queries of the geographic object data model to the geometric object data model and generates an optimized query processing plan. The function implementor then assigns the optimal routines to each of the functions and executes the query processing plan. It interacts with the physical database for the invocation of the precompiled routines of the functions stored in it and for the retrieval of the relevant objects. The spatial and non-spatial data together with their indices are also stored in the physical database.

3.1. Query Processor

The system accepts queries from the user through an interface language called Object-oriented Functional Query Language (OFQL) which will be discussed in Section 5.1. A user query is basically composed of a sequence of semantic spatial functions or predicates and/or other non-spatial functions or predicates. Upon receiving a query, the query processor checks the validity of each function and its arguments, after which, the query is transformed into a query processing graph where different abstraction hierarchies and optimization heuristics based on the database statistics are applied to generate the optimized query processing plan. The semantic spatial functions in the geographic object data model are mapped to their corresponding geometric functions or sequences of geometric functions in the geometric object data model. The non-spatial functions like retrieval functions and functions which make use of the abstraction hierarchies, on the other hand, need not be mapped. Thus, the optimized query processing plan generated by the query processor contains a sequence of geometric functions and/or non-spatial functions together with some control statements that specify the control mechanism or the actual sequence by which the execution of the query is to be done.

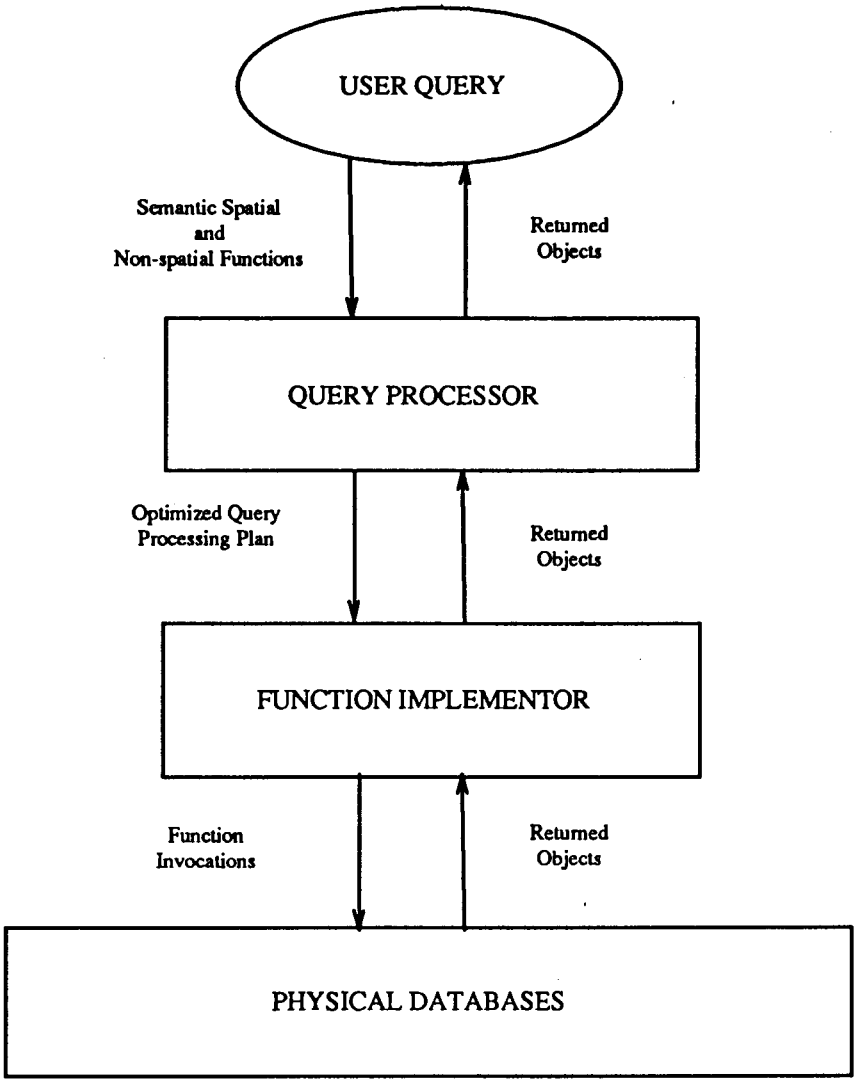


Figure 3-1: Architecture of a GIS System

3.2. Function Implementor

The function implementor is the lowest-level module in our architecture. The main input to this module is the optimized query processing plan generated by the query processor.

Geometric functions are implemented using algorithms which may or may not invoke some other simpler geometric functions. For any given functions, it is possible that there are different routines or algorithms that

are stored in the database that can carry out the operation and generate the same results. The logic behind having more than one algorithm for the same function is to allow the function implementor to decide which algorithm is to be used based on the size of the given data set or database statistics.

An example of this is the Length^{-1} function which is equivalent to the notion of buffer generation. Suppose we want to find if a given set of wells which are represented as nodes at the geometric object data model is located within 20 kilometers from the boundary of a given polygon representing a city, there are two possible implementation routines that can carry out the function depending on the size of the given WELL data set.

If the data set consists of very few instances, in this example, these elements are the wells, then we can use a straightforward algorithm like the length computation of a line joining the node representing the WELL object to a point on the edge of the polygon that represents the CITY object to determine if the node falls within the distance specified. If the data set is very large, it may then be more appropriate to generate the buffer polygon and perform a *Contains* function afterwards to check if the buffer polygon encloses all the given nodes.

Thus, depending on the various database statistics, the function implementor will assign the best algorithm that could carry out each of the functions in the query processing plan. After this, the query processing plan is executed by invoking the precompiled routines or algorithms that are stored in the database. The physical database upon the execution of the routine will return the intermediate results back to the function implementor. These intermediate results are usually pointers or object identifiers of the selected objects. The function implementor will then use them as inputs to the next function depending on the sequence control specified in the query processing plan and the next function will then be invoked in the database. Finally, after the execution of the entire query processing plan, the result is retrieved from the database and presented to the user.

If there is only one algorithm that is precompiled and stored in the database for carrying out each function, then the function implementor could simply do static mapping to the routines in the database without any need of intelligence.

In this architecture, the basic routines for the actual retrieval, manipulation and computation of objects, together with the precompiled routines of the user-defined superfunctions are stored in the database. Superfunctions are discussed in Section 4.1.3. They are written using the functions provided in the data model to build more application-dependent functions that could be readily used by the high-level user. Note

that for the case of compiling superfunctions, it is similar to macro-expansions since upon being processed by the query processor, the superfunctions aside from having some additional computation and control statements, they are basically defined in terms of geographic and geometric functions which have their corresponding routines stored in the database.

As can be seen, behavioral abstraction is highly supported in our data model. Each layer of our bi-level data model could be mapped into the next layer with each layer working in its own premises, meaning, semantic functions on geographic objects, and geometric functions on geometric objects. Any changes, whether they be the algorithms used for the implementation of the geometric functions or changes in the storage representation of the spatial objects at the geometric object data model, will not affect the geographic object data model since the semantics of both the geographic objects and semantic functions are kept intact. In effect, each layer can function independently.

Chapter 4
Bi-Level Data Model

There are two separate layers in our object-oriented data model. They are the geographic object data model, and the geometric object data model which are discussed in Section 4.1 and 4.2 respectively. Please refer to Figure 4-1.

4.1. Geographic Object Data Model

The geographic object data model primarily consists of geographic objects and a set of semantic spatial functions. With the geographic objects, the details of the geographic objects, e.g., spatial representation, are transparent to the user. For example, the user need not be aware of whether they are stored in raster format or vector format. User can perform spatial and non-spatial queries by directly referencing these geographic objects using OFQL. We will discuss objects, different types of abstraction hierarchies and functions that are found in this geographic object data model in Section 4.1.1, 4.1.2 and 4.1.3 respectively.

4.1.1. Objects

At this geographic object data model, objects could be generally classified into geographic objects which are the non-primitive objects and the primitive objects.

Each distinct type of physical feature found on the earth's surface including the distinct themes portrayed in the different map layers makes up a distinct object class with each of its instances representing a specific geographic object. Geographic objects could be defined with some intrinsic properties or attributes like Name, Population and others which could be accessed using retrieval functions. They could be either persistent or transient. By persistent geographic objects, we refer to those which are stored at the initial creation of the database and those query-generated objects which are later on explicitly specified for storage by the user. The transient objects, on the other hand, are the resulting objects generated for the user queries which exist only throughout a user session, and could be referenced by any user queries in the same session. They are lost upon termination of a user session since they are not stored in the database.

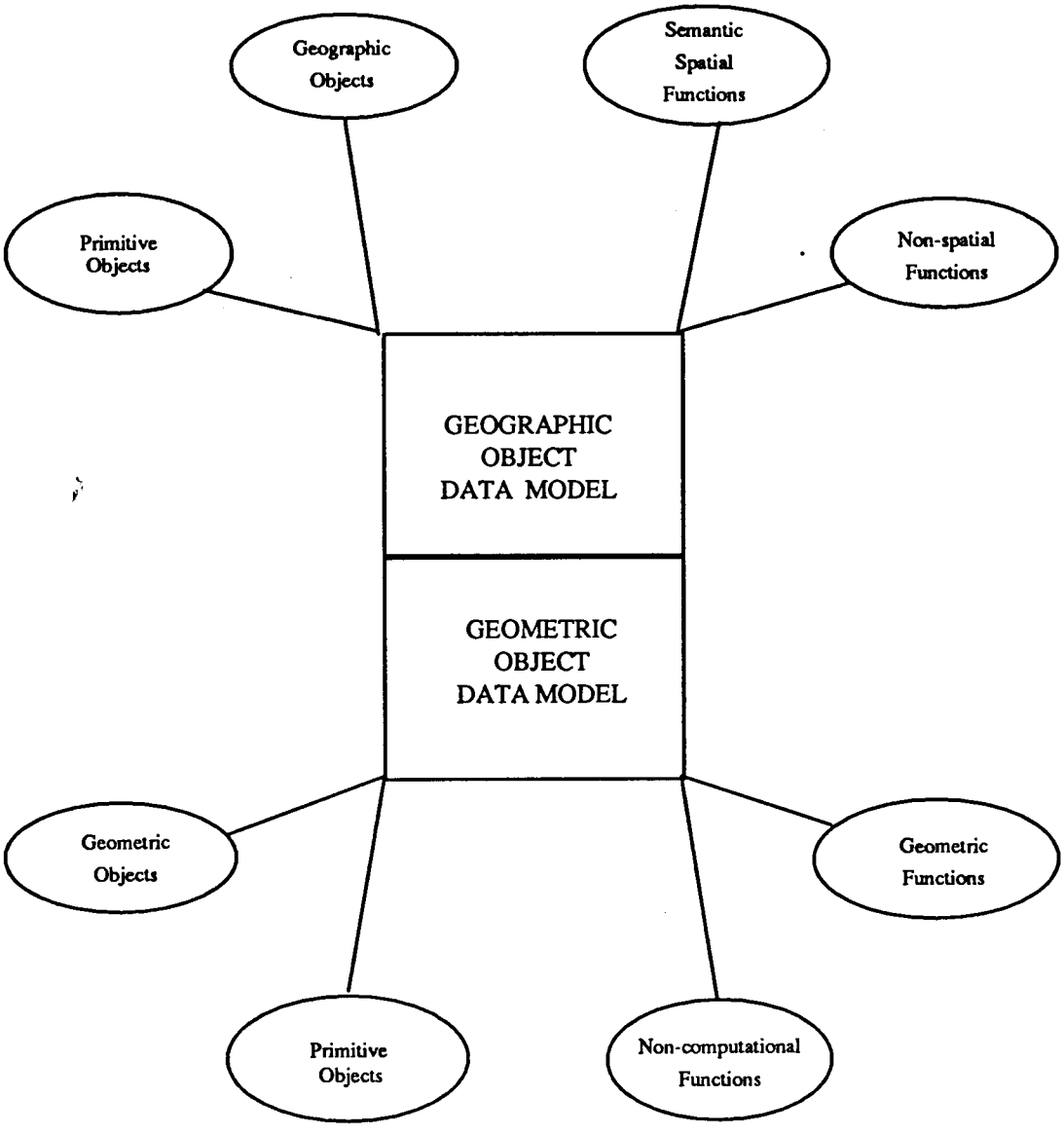


Figure 4-1: Bi-Level Object-Oriented Data Model for GIS

Examples of objects which may be generated at run-time are path (which consists of a sequence of strips that connect two physical points on the earth's surface), and a physical region which satisfies several conditions based on the different themes, e.g., a commercial zone of a city that has unstable soil.

Each geographic object has a unique object identifier which is actually a pointer and has its corresponding *distinct* geometric representation at the geometric object data model if it is *spatially-associated.* The latter clause will be explained and clarified in the discussion of IS-A abstraction in the next section.

The primitive objects as briefly discussed in Section 2.2.1, also have unique object identifiers and are of built-in system types. They could also be either persistent or transient just like the geographic objects.

4.1.2. Types of Hierarchies

We have generally two types of abstraction hierarchy in the geographic object data model, namely: PART-OF and IS-A abstractions. The PART-OF abstraction models the total containment relationship among objects. To distinguish the PART-OF hierarchy from the IS-A hierarchy, we will use the term *aggregate parent object* to refer to the object which is made up of some constituent component objects related to it by the PART-OF modeling abstraction and *superclass object* to refer to the object which is a generalization of some other subclass objects related to it via the IS-A modeling abstraction. Generally, the PART-OF hierarchy is organized based on the spatial relationships among geographic objects, while the IS-A hierarchy is created based on the non-spatial properties of objects. We will discuss these abstractions in the rest of this section.

First, let us discuss about the *PART-OF* or the *containment-part-of* abstraction. In a PART-OF hierarchy, the component objects, which are usually of different object classes, make up an aggregate parent object. It models the containment relationship of the geographic objects that define the surface content of the aggregate parent geographic object. The different component objects may or may not overlap with one another spatially. Please refer to Figure 4-2 for an example of a PART-OF hierarchy where the component objects do not overlap each other spatially.

Here, an airport consists of a terminal building, a runway, and a control tower. In this case, the aggregate parent object class is defined as an aggregate of *all* the component object classes at any point in time. TERMINALBLDG alone could not make up an airport. The component objects belong to distinct object classes and have their own sets of functions which are most possibly different from those found in the aggregate parent object class. There is no inheritance of functions from the aggregate parent object class to the component object classes. The PART-OF abstraction is specified in the object class definition using the *PART-OF* keyword as in the sample TERMINALBLDG object class definition below:

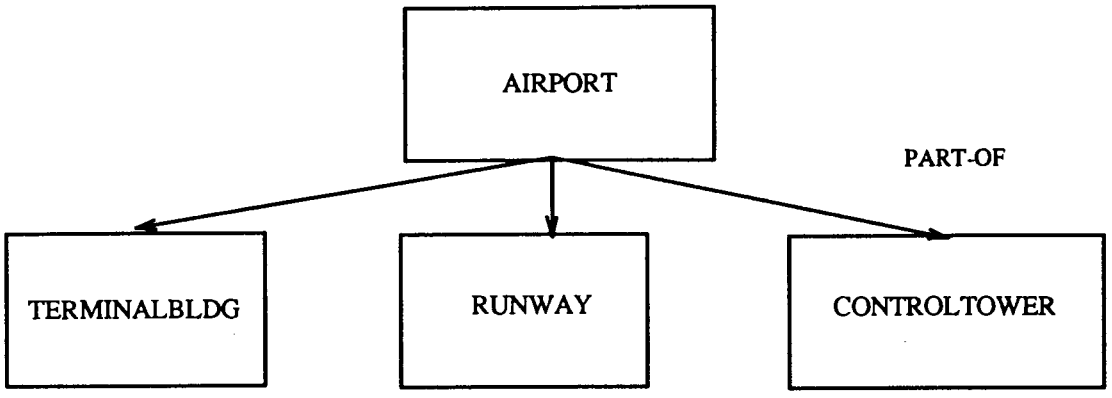


Figure 4-2: PART-OF Abstraction for the AIRPORT Object

create objectclass TERMINALBLDG
PART-OF : AIRPORT
IS-A : BUILDING
Year : INTEGER
Type : STRING

The IS-A keyword will be discussed in the IS-A abstraction later on. Year and Type are functions defined for the TERMINALBLDG object class which map to some primitive objects of data type INTEGER and STRING respectively.

To illustrate an example wherein the surface content of an aggregate parent object could be completely defined spatially by *any* of its component object classes is shown in Figure 4-3.

In the figure, a COUNTRY object could be spatially defined in terms of all its PROVINCE objects. Each PROVINCE object, in turn, consists of CITY objects, and each CITY object is defined spatially in terms of *either* all its TOWN objects or all its DISTRICT objects. To look at the other branch of the hierarchy, a COUNTRY object could also be viewed as spatially defined in terms of all its REGION objects, each of which consists of CITY objects. This PART-OF hierarchy could be used in modeling containment relationships among geographic objects that are classified under the political, administrative and economic boundaries or groupings. As can be seen from the example above, the component object classes that are directly connected to the same aggregate parent object via the PART-OF modeling abstraction may overlap one another spatially. The PROVINCE and CITY object class definition is presented below.

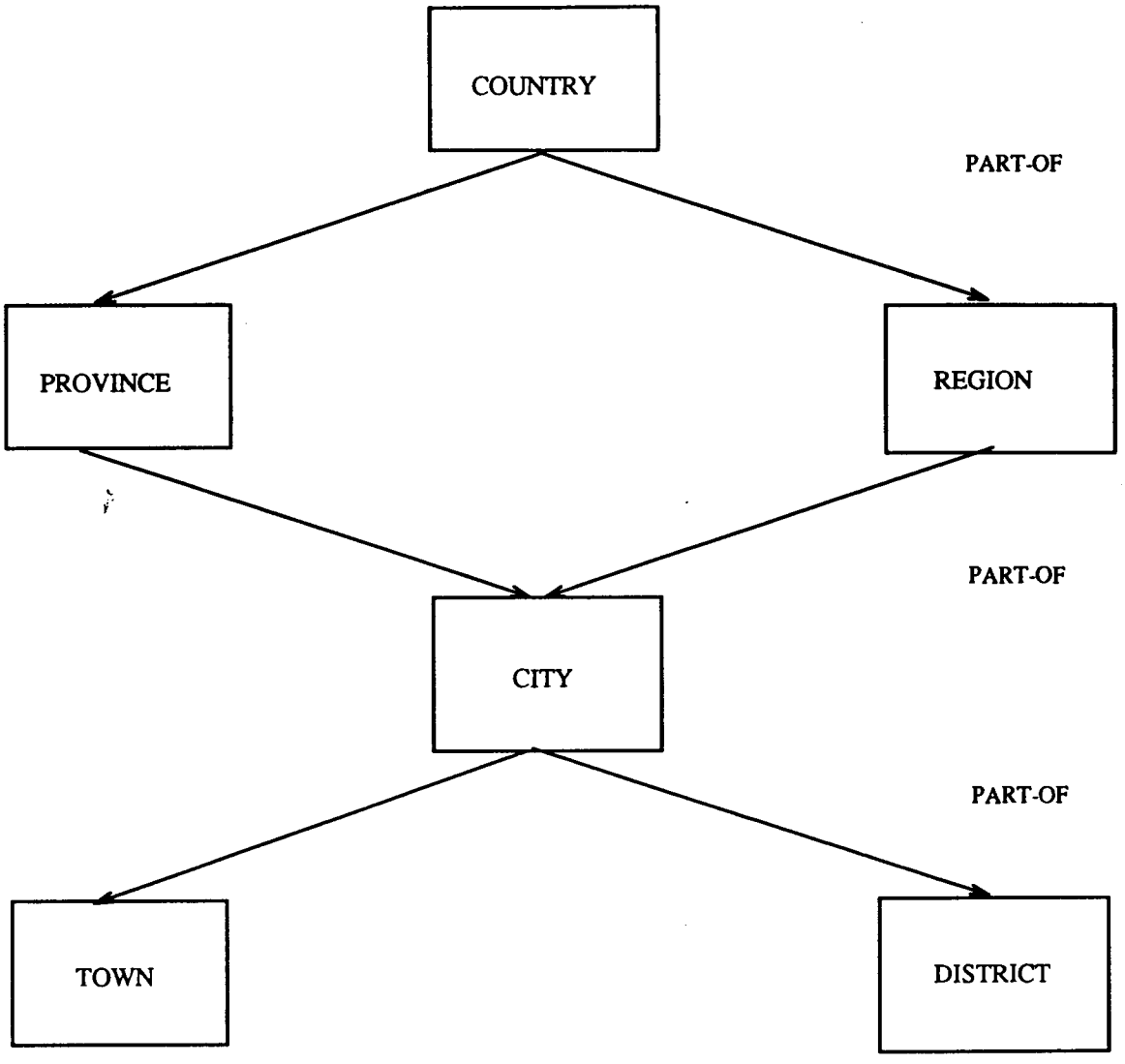


Figure 4-3: PART-OF Abstraction for the COUNTRY Object

create objectclass PROVINCE
PART-OF: COUNTRY
Population: Sum(Population of CITY)

create objectclass CITY
PART-OF: PROVINCE
PART-OF: REGION
Population: INTEGER

It is obvious in the description of the PART-OF hierarchy that the deduction of containment is directly derived among two objects if and only if they could be found along the same one-way path. A *one-way path*

exists between two object classes if they can reach one another by traversing strictly in either a top-down or bottom-up direction of the PART-OF hierarchy. Using Figure 4-3 as an example, we cannot figure out if a town overlaps a certain district nor can we find out if a province overlaps a particular region since they are connected to the same aggregate parent object and by no means could we find a one-way path that connects the two.

The advantage of using a PART-OF abstraction hierarchy is that an object has *direct access* to its aggregate parent objects and/or all its component objects. The pointers to these objects should be *labeled* to distinguish aggregate and/or component objects of the different object classes.

Let us consider a COUNTRY object as an example. Referring to Figure 4-3, a COUNTRY object is composed of a specific set of PROVINCE objects or it could also be seen as a composition of some REGION objects. The COUNTRY object has labeled pointers directed towards all the PROVINCE objects and REGION objects which are totally contained in it. Thus, once we have access to a COUNTRY object, using these labeled pointers or links, we can directly access the specific PROVINCE objects or REGION objects that are *part-of* this COUNTRY object.

The PART-OF abstraction supports *upward propagation* which could be classified as either *selective upward aggregation* or *upward containment transitivity*. An example of selective upward aggregation could be found in the sample COUNTRY object class definition of Section 2.2.2. We will rewrite the *Population* function in the COUNTRY object class here.

Population: Sum(Population of PROVINCE)

What it means here is that for each COUNTRY object, use the PART-OF labeled pointers to access all the PROVINCE objects that are contained in it and get the sum of the population of this relevant set of PROVINCE objects. Looking at the previous object class definition of PROVINCE and CITY in this section, the population of each PROVINCE is determined to be the sum of the population of the CITY objects that are PART-OF related to it. Thus, what we need is to follow the PART-OF pointers labeled as CITY of each of these selected PROVINCE objects to access the specific CITY objects which are contained in each of the relevant provinces. After that, the sum of the population of all the CITY objects of each of the previously selected PROVINCE objects is obtained and becomes the population of the respective PROVINCE object. Upon getting the population of all these selected PROVINCE objects, their population values are added together and propagated back to the given COUNTRY object instance as its population value.

As can be seen, this PART-OF hierarchy facilitates direct access of the aggregate parent object to only the relevant component objects and the functions of the aggregate parent object could also be directly defined in terms of the functions in its component objects. This entire process illustrates how upward aggregation is done. We call it selective because not all functions in the component object classes are needed by the aggregate parent object class to define its own functions. It all depends on the application requirement to select only those functions which are relevant and are needed by the aggregate parent object.

In other words, function of an object at a certain level of the PART-OF hierarchy could be defined as aggregation or other mathematical computations such as average on the results mapped by the specified functions defined in a component object class that is directly or indirectly connected to it via the PART-OF modeling abstraction. Again, we should make sure that for selective upward aggregation to be applicable, these two object classes must be found along the same one-way path.

Upward containment transitivity means that if an object at level i contains an object a, then its aggregate parent object, which is at the $(i-1)^{th}$ level, will also contain the object a. Notice that the reverse is not always true.

Using Figure 4-3 again as an illustration, if a CITY object contains TOWN object a, then the PROVINCE and COUNTRY objects of which the CITY object is PART-OF related to them also contain that TOWN object a; but if a COUNTRY object totally contains DISTRICT b, it does not mean that all the CITY objects that are PART-OF related to that COUNTRY object also totally contain DISTRICT b. We would like to repeat that upward containment transitivity is applicable only between object classes that can be found along the same one-way path in the PART-OF hierarchy.

With the use of the PART-OF abstraction, query processor needs not consult spatial indices nor needs any geometric computations to figure out the containment relationship between two geographic objects situated along the same one-way path of the PART-OF hierarchy, thus, giving immediate results to the containment queries.

Now, we are going to discuss about generalization or the IS-A class hierarchy. We would like to show that IS-A abstraction is also an important modeling concept in GIS applications. IS-A class hierarchy allows the inheritance of functions from the superclass to its subclasses. An example of this is shown in the previous TERMINALBLDG object class definition in the early part of this section which specifies that a terminal building is a kind of BUILDING and it inherits all the functions of the BUILDING object class in addition to its own set of functions.

An IS-A class hierarchy with WATERBODY as the superclass and STREAM, RIVER, SEA, LAKE and WATERFALL as its subclasses is shown in Figure 4-4. Here, we have disjoint classification wherein each WATERBODY object can only be classified as an instance of at most one of its subclasses. It is impossible for us to find any of the subclass instances that possesses the same object identifier. Any instances in the respective subclasses of the WATERBODY superclass may play different roles depending on their usage. For example, they could be used as a national boundary, a utility source, a drainage channel, a transport route, and/or for recreational purposes. Thus, when the user is interested in retrieving all drainage channels, the system will retrieve all the water bodies with "drainage" as the value that is mapped to it by the *Usage* function defined in these subclass definitions.

Multiple inheritance is also applicable to GIS applications wherein an object class may have more than one superclass. An example of its use is in modeling the various *themes*. If a river has double usage, which means it participates in several themes, then it may have several superclasses. For example, a RIVER object that is used for recreational purposes and is also used as a transport route will have two other superclasses, RECREATIONAL_WATER and WATER_TRANSPORT_ROUTE object classes, aside from the WATERBODY object class. RECREATIONAL_WATER object class consists of all types of water bodies that are used for recreational purposes and it may also be a subclass of still another object class like the RECREATIONAL_SPOT object class, thus, creating an IS-A class hierarchy.

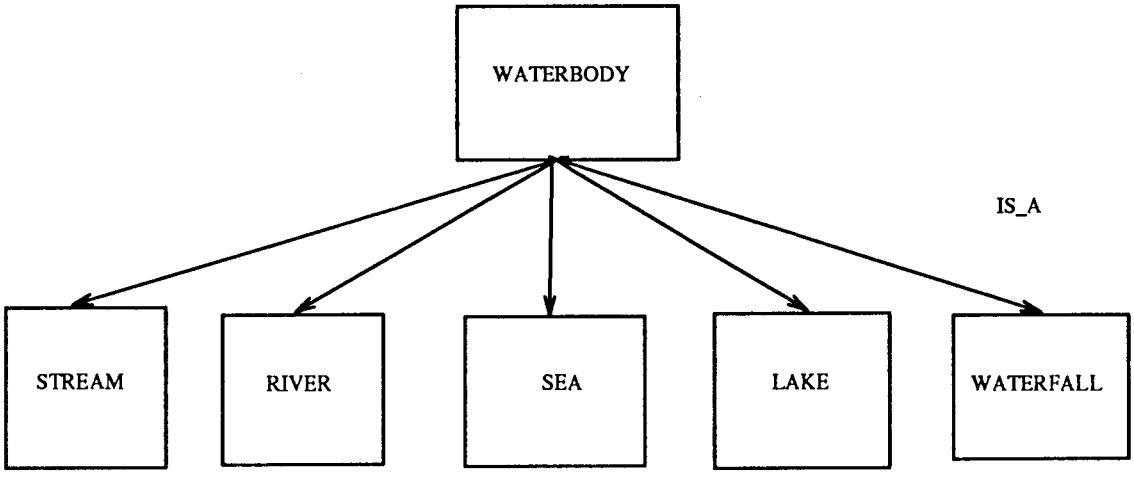


Figure 4-4: IS-A Class Hierarchy of the WATERBODY Object Class

To give a glimpse of how multiple inheritance could be handled, we will just point out some of the existing methods discussed in literatures. The first one is to have the subclass inherit its functions from the *immediate superclass* which is the first superclass in the list of superclasses. Another method will be the assignment of new function names in the subclass to refer to the function from different superclasses. Finally, the user could also explicitly specify from which superclass it is to inherit. We will not be discussing how multiple inheritance is handled in detail since it is discussed in many other literatures [Banerjee et al. 87], [Rowe 87]. Note that for our case, inheritance will not be limited to attributes only, but is also applicable to other functions in our data model including the semantic spatial functions.

We can look at the IS-A abstraction hierarchy as a directory. Only the distinct object classes in the *leaf nodes* of an IS-A class hierarchy have direct geometric representations in the geometric object data model, and these object classes in the geographic object data model are called *spatially-associated geographic object classes.* Those object classes which are at the non-leaf nodes of an IS-A class hierarchy will not have any geometric representations directly associated with them in the geometric object data model. Instead, they are directories which point to all their subclasses. These subclasses may again be directories or they could have instances that have direct geometric representations at the geometric object data model. Thus, any spatial functions performed on the superclass will imply the traversal from the superclass to its subclasses which instances have direct geometric representations in the geometric object data model. Suppose a RIVER object has its own geometric representation and it is a subclass object of both the RECREATIONAL_WATER and WATER_TRANSPORT_ROUTE object classes, this RIVER object will not have three distinct geometric representations to represent its instantiation in the three object classes, RIVER, RECREATIONAL_WATER and WATER_TRANSPORT_ROUTE respectively. There will only be one geometric representation in the geometric object data model that represents it since its instantiation in these three object classes refer to that same RIVER object. Thus, we can see that geographic object classes that are at the non-leaf nodes in an IS-A class hierarchy are implemented as directories which do not have a distinct and *direct* mapping to unique geometric objects in the geometric object data model. They are called the *non-spatially-associated geographic object classes.*

4.1.3. Functions

There are several types of functions in the geographic object data model. They include the semantic spatial functions, retrieval functions, set functions, aggregate functions, inverse functions and superfunctions. All these functions could be used in writing user queries using our query language OFQL.

The **semantic spatial functions** are the spatial operations specified by the user in the user query as an interface to the system to retrieve, manipulate or compute for geographic objects. The main idea behind semantic spatial functions is to provide a set of tools or spatial functions to the user to deal with the geographic objects, and from which more complicated queries could be built. Semantic spatial functions simply abstract the underlying geometric representation and computation of the geographic objects from the user. Below are some of the suggested semantic spatial functions, and by no means are the following functions complete. The designer of the database model can define his own set of semantic spatial functions which provides the building blocks for the user to write more sophisticated queries.

Eight sample semantic spatial functions are provided in this thesis as follows:

Adjacent, Area, Complement, Contains, Intersects, Length, Route, and *Surrounds.*

Function overloading is allowed wherein the same function name may mean different semantics depending on the object class of its I/O arguments, and will thus, produce different results. Below is an example of the *Intersects* function which illustrates the different semantics it has depending on the object classes for its input and output parameters.

- Intersects (INTERSECTION) -> STREET

  Find the respective sets of STREET objects that intersect the individual INTERSECTION objects in the given input set.

- Intersects (STREET) -> HIGHWAY

  Find all the respective sets of HIGHWAY objects that intersect the individual STREET objects in the given input set.

  In the geometric object data model, this refers to as the edge-edge intersection.

- Intersects (FOREST) -> RAILWAY

  Find all the respective sets of RAILWAY objects that intersect with the individual FOREST objects in the given input set.

  In the geometric object data model, this refers to as the polygon-edge intersection.

- Intersects (FOREST, SOIL) -> LAND

  Find the query-generated LAND objects where the FOREST and SOIL objects overlap. Note that the FOREST and SOIL objects here may not be involving all the instances of the entire FOREST and SOIL object classes. The input SOIL objects to the function may be qualified to soil of loam type, for example, and the FOREST objects here may just be a set of evergreen forests. In this case, we are looking for the exact land areas where there are loam soil and evergreen forests.

  In the geometric object data model, this may refer to as the polygon-polygon intersection.

Retrieval functions include the functions for retrieving properties or attributes, traversing the different abstraction hierarchies of objects (i.e., PART-OF, and IS-A hierarchies), and retrieving the explicitly stored relationships among objects which are seen as mappings from an object to other object(s). Below are some examples for each of the above types of retrieval functions respectively:

- Population(CITY) -> INTEGER
- Contains(COUNTRY) -> CITY

  The PART-OF hierarchy is used to generate the answer for this function.

- OwnedBy(LOT) -> PERSON

The **set functions** may include the following:

- Count: It counts the number of elements in a set.
- Union: It merges or gets the union of several sets into a single set of elements. If the input arguments to the Union function are sets, they are unnested and will be merged to form a single set of non-set elements.
- Common: It returns the elements that are found in all the sets specified as input arguments to this function.
- Distinct: It eliminates duplicate elements in a set. The result of this function is a set of distinct objects.

The **aggregate functions** include the following:

- Minimum: It returns the minimum value among all elements in a given set.
- Maximum: It returns the maximum value among all elements in a given set.
- Sum: It returns the sum of all the elements in a given set.
- Average: It returns the average of all the elements in a given set.
- Percentage: It returns the percentage of an element over the entire set of elements.

The concept of **inverse functions** is very crucial especially when we discuss query processing later on in Chapter 6. Let A and B be two object classes. Let *a* represent an instance from object class A and *b* represent a set of instances from object class B which is mapped to it by a function F with *a* as the input argument. We could use F(A)->B to symbolize the mapping association between object classes A and B via the function F. If each of the object class instances in set *b* could also be mapped to the corresponding object class instance *a* in F(A)->B by function F^{-1}, then F^{-1} is said to be the **inverse function** of F and vice versa. In other words, given the functions F(A)->B and F^{-1}(B)->A where all pairs of *a* and *b* instances produced as a result of mapping among the two object classes A and B using the function F are exactly the same as that produced by the mapping of the function F^{-1} and vice versa, then the functions F and F^{-1} are said to be *inverse* of each other.

We say that function F in the mapping of F(A) -> B has a corresponding inverse function, F^{-1}, for the mapping of F^{-1}(B) -> A if the domains of A and B are finite. This statement is valid because if an *a* instance in object class A could be mapped to a finite set of *b* instances, say, $\{b_1,..., b_n\}$ in object class B, then *n* tuples will be produced for the resulting finite set relation of F(A) -> B considering one particular *a* instance only where each tuple could be represented as $<a, b_i>$ where *i* goes from 1 to n. Thus, if domain A is finite, then it will produce a finite set relation, F(A) -> B, where each tuple consists of an *a* instance and a *b* instance.

As can be seen, F(A) -> B can be considered as a virtual relation (relation in the relational database) where each tuple contains an *a* instance and a *b* instance provided that A and B are finite. In our case, this relation realization of a function is stored as tuple objects. The reasoning for F^{-1}(B) -> A is the same. Since B is a finite domain, it will be mapped to a finite set of *a* instances in A. There is a mapping of *b* instance to some finite set of *a* instances in A if it is found in the tuples from the resulting finite set relation generated by F.

If the input argument of a function is of primitive type, then it must be instantiated, otherwise, the execution of that function may possibly end up in an infinite loop since unlike non-primitive object classes, the number of instances in a primitive object class is limited only by the system. The built-in type, INTEGER, for example, may have its number of instances limited only by the number of bits used by the system in representing an integer.

Please refer to the following for some examples of inverse functions.

Example:

- Contains(COUNTRY) -> FOREST

  $Contains^{-1}$(FOREST) -> COUNTRY

  Here, $Contains^{-1}$ is semantically equivalent to EnclosedBy which is actually the inverse of *Contains*. If COUNTRY *a* contains a set of ten FOREST objects *b*, then each of these ten FOREST objects *b* is also enclosed by the COUNTRY *a*. Since we have a finite number of instances for object classes COUNTRY and FOREST, so the mapping from the entire object class COUNTRY to object class FOREST via the *Contains* function will be the same as the mapping from the entire object class FOREST to object class COUNTRY via its inverse function $Contains^{-1}$.

- Age(PERSON) -> 20

  Age^{-1}(20) -> PERSON

  Age, here, is a retrieval function which models an attribute or property of the object class PERSON. Each PERSON object is mapped to a certain integer via the function *Age*. The inverse function Age^{-1} will generate all the PERSON objects having a particular age. Thus, if there are ten instances in the entire object class PERSON which are mapped to the integer 20 via the function *Age*, then the integer 20 will be mapped to these ten PERSON objects via the inverse function Age^{-1}.

Inverse function holds also for functions which involve several object classes for their input and/or output parameters provided that the domains of the parameters are finite. For example, Distance(C1, C2) -> REAL. C1 and C2 are aliases for the CITY object class. This function wants to find the distance between two CITY objects at any given point of time wherein the output is of REAL data type. Suppose we have the inverse

function Distance^{-1}(C1, REAL) -> C2, this is valid only if the REAL domain is instantiated prior to the execution of this function, otherwise, the REAL object class itself is not finite. This function returns the C2 objects which are separated from the C1 objects with a distance that is specified in the REAL number of the input parameter of the function. Suppose, instead of REAL, we have a range of REAL numbers specified like <=REAL as the input parameter. C2 objects in the original mapping Distance(C1, C2) -> REAL will be returned as results if there is a C1 object in the resulting finite set relation represented as Distance(C1, C2) -> REAL and its corresponding REAL object associated with it is less than or equal to the specified REAL number.

Superfunctions are user-defined procedures that may invoke functions, nested functions, and some other additional computations or routines (which include a set of arithmetic and logical operators and control statements) to provide system extensibility to cater for the needs of the different GIS domain-specific applications. Usually, superfunctions are not defined by end-user, but by a group of programmers who provides different add-on functions for the different application domains.

We include nested function in defining a superfunction because it provides a more natural and straightforward way of expressing a computation. The intermediate results of an inner component function are interpreted as automatic inputs to the next outer component function of the nested function. Each of these component functions in a nested function can be any of the five other types of functions (i.e., spatial functions which include both semantic spatial functions and geometric functions, retrieval functions, set functions, aggregate functions and inverse functions) or other previously defined superfunctions. We need geometric functions aside from the functions in the geographic object data model in defining superfunctions for some applications that need to deal with data from both layers of the bi-level data model in the computation of the final result. Finding the localities that are within a driving distance of 50 miles given a specific starting point or finding the optimal driving route between two places considering all the parameters like traffic, road conditions, weather, and others are some of the practical examples.

Below is a very simple example of the ShortestRoute superfunction definition which is defined using nested functions. ShortestRoute superfunction calculates for the shortest distance between any two cities.

```
C1 alias CITY
C2 alias CITY
define ShortestRoute(C1, C2) -> INTEGER
as
  select INTEGER
  where
     Minimum(
          Length(
               Route(C1, C2) -> PATH
             ) -> INTEGER
          ) -> INTEGER
```

Evaluation of nested functions starts from the innermost component function. To explain the functions specified in the superfunction above, we have the following:

- Route(C1, C2) -> PATH

  Given two CITY objects of object classes C1 and C2 respectively, generate the different PATH objects that can connect the two together.

- Length(PATH) -> INTEGER

  Using the unified PATH objects as a result of evaluating Route(C1, C2) -> PATH, the Length function is invoked to map the PATH objects to their corresponding length which is of type INTEGER.

- Minimum(INTEGER) -> INTEGER

  The INTEGER instances used as inputs to the Minimum function represent the length of the paths as were generated in the previous function. The output INTEGER instance is the minimum INTEGER object among those input INTEGER objects of the function.

The user could invoke superfunctions in his queries just like using any other functions. Thus, previous types of functions in the geographic object data model, together with the superfunctions appear the same to the end-user as a custom-made toolbox of functions that can be used for his particular application. This pool of functions can be further expanded by defining more superfunctions in terms of the existing set of functions. A few examples of these are *Closest, ShortestDrivingDistance, ShortestWalkingDistance, OptimalPath* and others.

4.2. Geometric Object Data Model

The geometric object data model primarily includes geometric objects or the actual spatial representation of all the geographic objects and a set of geometric functions. Objects and geometric functions will be discussed in Section 4.2.1 and 4.2.2 respectively.

4.2.1. Objects

There are two general types of objects in the geometric object data model. They are the geometric objects, and the primitive objects. Similar to the objects in the geographic object data model, they could be persistent or transient and each of them has a unique object identifier.

Each spatially-associated geographic object in the geographic object data model has its corresponding *distinct* geometric objects at the geometric object data model. Several geometric objects having exactly the same coordinates may be produced to represent different spatially-associated geographic objects which happen to occupy the same physical space on earth's surface. An example of this is the FOREST object which has loam soil type and it just happens that the physical area where there is loam soil totally coincides with the boundary of the forest.

We use vector representation for the spatial representation of our geographic objects. Based on [Star and Estes 90], the geometric object classes that we have in this data model are POINT, LINE, NODE, POLYGON and CHAIN. A *point* has no area and is often represented as a pair of numbers like latitude-longitude, or x-y coordinates. A *line* is made up of a connected sequence of points with a starting node and an ending node. *Node* is a special type of point which indicates junctions between lines or the ends of a line segment. *Polygon* is defined by a set of line segments called *chains* which collectively defines its boundaries. We allow the sharing of chains among different geometric objects if the user prefers to. For example, if a river is used to define the national boundary, and there are two distinct polygons in the geometric object data model that represent the RIVER and the TOWN objects, each having different object-identifiers respectively, then the chain of the polygon representing the RIVER object that is common to the chain of the polygon representing the TOWN object will be shared. In this case, if ever the chain that spatially defines the polygon representing the RIVER object changes, then the boundary chain of the polygon representing the TOWN object will also be changed automatically and vice versa since both of them share the same chain which means that they have the same object-identifier for that CHAIN object. The advantage of this is not only to eliminate data redundancy, but most of all, it frees the user from having to explicitly

update different objects once the boundary changes. Thus, if a chain is shared by several geometric objects, then only one update is needed to make the data consistent.

Primitive objects are objects which are of built-in system types like INTEGER, STRING, REAL, RANGE and BOOLEAN.

The query-generated geometric objects which are the spatial representation of geographic objects derived from the query are generated at run-time as a result of spatial computations. For example, the spatial *CommonPoly* function will produce a new POLYGON object which represents the portion where the two given POLYGON objects overlap.

4.2.2. Functions

All the functions in the geographic object data model could be used in the geometric object data model except for the semantic spatial functions which are mapped by the query processor to a set of geometric functions in the geometric object data model.

The *Geometric functions* are the functions that perform spatial overlays, containment, intersection and other spatial searches, area and length computations.

The implementation of a geometric function is a routine which may or may not invoke some other geometric functions. Since geometric functions depend on the underlying spatial representation of the geographic objects, we will just mention some of the possible geometric functions that could be used to manipulate or compute for the geometric objects in vector representation. They are grouped according to the similarity of functions that they perform. There is no way by which they are complete. Depending on the algorithms of computation, some other geometric functions can be added or some geometric functions from the specified groups below could be eliminated.

The five groups of geometric functions are overlap or intersect operations, containment operations, component or aggregate operations, geometric object computations, and arithmetic computations on attributes of geometric objects.

Overlap or *intersect operations* generally perform polygon overlaps, and intersections among lines, chains, points and nodes. They include *FindOverlap, CommonPoly, ComplementPoly, IntersectingNode, Lie*, and *CommonChain*. Below is an example for the function ComplementPoly:

ComplementPoly (P1, P2) -> P3

P1, P2 and P3 are aliases for the POLYGON object class. Aliases will be discussed in Section 5.1. Here, given two subsets of POLYGON objects P1 and P2, *ComplementPoly* generates the query-generated POLYGON objects P3 which represent the portions that are not overlapped by the given POLYGON objects in the input parameter list of the function.

Containment operations perform containment and enclosure operations on polygons, lines, chains, points and nodes. It includes the function *Contains*.

Component or aggregate operations retrieve the components or the aggregate of a geometric object. For example, they could find the chains that define a polygon, the line segments of a line, the set of lines or line segments that has a given node as either its starting or ending node, the intermediate nodes of a line, and the line of a given line segment. The functions defined under this group include: *FindChains*, *ComponentSegments*, *IntermediateNodes* and *AggregateLine*.

Geometric object computations may include the function *Centroid* which calculates for the centroid of a given polygon and the function *Line* which creates a LINE object that joins between two points.

Arithmetic computations on attributes of geometric objects include functions *CalculateLength* and *Area* which calculate for the length of a line or chain and the area of a polygon respectively.

Chapter 5
System Interfaces

5.1. OFQL: Object-oriented Functional Query Language

OFQL is the main user interface of our object-oriented data model used for retrieval, manipulation, and computation which could answer spatial and non-spatial queries of a geographic nature. It is used in posing ad hoc queries for different what-if scenarios which are crucial in GIS applications wherein the user needs to specify different conditions that have to be met for a geographic object to be selected. Details of spatial representation and computation of geographic objects are transparent to the user. The actual representation of the spatial data is totally abstracted from the user at the user-query level.

OFQL has a predicate format which uses the functions introduced in Section 4.1.3. It is declarative in both form and nature and is easy to learn for even the user with little computer experience. An OFQL user query is specified in a SELECT-WHERE block. The statement after the SELECT clause specifies the **target object classes** which objects that satisfy all the predicates specified after the WHERE clause are returned as results to the query.

Predicate which is used to define the body of the WHERE clause of an OFQL query includes both the input and output parameters of any of the semantic spatial functions, retrieval functions, superfunctions, set functions, aggregate functions and/or inverse functions in its parameter list which eventually yields a boolean result, TRUE or FALSE. The predicate is evaluated to true if there exists a non-null value or object that could be bound to each of the parameters to make the predicate true.

The general format of predicate is written as *function-name(parameter list)*. The position of these parameters in the parameter list of a predicate has a one-to-one correspondence to the I/O parameter specification of its corresponding function.

For functions involving objects of built-in data types as their parameters, comparators like =, <, >, <>, <=, >= could be specified before the primitive object class names or primitive objects. By default, = is assumed to be the comparator used.

A simple OFQL query which uses both semantic spatial predicates and non-spatial predicates is shown in the example below.

Example 5-1. Find the recreational lakes that are enclosed in an evergreen forest and are adjacent to a road.

```
select LAKE
where
      Usage(LAKE, "recreational")
   And
      Contains(FOREST, LAKE)
   And
      Type(FOREST, "evergreen")
   And
      Adjacent(LAKE, ROAD)
```

In this example, each of the predicates are connected with the logical AND connective. Thus, in order for an instance from the LAKE object class (which is specified after the SELECT clause) to be selected, there should exist a FOREST object and a ROAD object that could be mapped to the LAKE instance and could satisfy all the predicates of the query. If a forest contains a certain lake, then this forest must be of evergreen type, and the generated LAKE object must be adjacent to a road and is of recreational usage in order for this LAKE object to be returned as one of the results of this query.

Aliases are used to distinguish two different subsets of objects from the same object class which may have distinct characteristics. They exist only throughout the lifetime of an user query.

Example 5-2. Find all roads that intersect with the roads which are adjacent to the university "SFU" or "UBC".

```
R1 alias ROAD
R2 alias ROAD
select R1
where
      Intersects(R1, R2)
   And
      (
        Adjacent(R2, UNIVERSITY)
      And
        Name(UNIVERSITY, "SFU")
   Or
        Adjacent(R2, UNIVERSITY)
      And
        Name(UNIVERSITY, "UBC")
      )
```

In the example above, R1 is used as an alias for the subset of ROAD objects which satisfy all the predicates of the query. R2 is an alias representing the subset of instances from the ROAD object class which are adjacent to either "SFU" or "UBC". Since R1 and R2 are of the same object class, ROAD, and are used in the same query, it is important for us to distinguish between the two by using aliases when we are particular as to which of them we are referring to.

In relation to the query above, we can see that parentheses can be used to specify the grouping among functions instead of the default interpretation that AND has a higher priority than OR when grouping functions. Notice that the priority levels are important to remove ambiguities in the interpretation of queries. In that example, the sequence by which the predicates are interpreted is shown below in descending priority.

- The two items below are of the same priority level.
  - Adjacent(R2, UNIVERSITY)
    And
    Name(UNIVERSITY, "SFU")
  - Adjacent(R2, UNIVERSITY)
    And
    Name(UNIVERSITY, "UBC")
- The OR logical connective has the next highest priority level due to the parentheses.
- The AND logical connective which connects the predicate Intersects(R1, R2) with the query segment in parentheses has the lowest priority in this example.

All predicates in a user query which have a certain object class name, say A, as one of the parameters in their respective parameter lists, and if these predicates are connected to each other via the AND logical connective, then this implies a stronger constraint on the set of instances from object class A that can be selected as relevant to the query. Let us refer to the example below for a brief illustration.

Example 5-3. Find the evergreen forests of the country "India" which are used for commercial purposes and which contain a lake in them.

```
select FOREST
where
    Name(COUNTRY, "India")
And
    Contains(COUNTRY, FOREST)
And
    Type(FOREST, "evergreen")
And
    Usage(FOREST, "commercial")
And
    Contains(FOREST, LAKE)
```

The predicates which have FOREST as one of their parameters are the two *Contains* functions, *Type* and *Usage* functions. Assuming the predicates are executed in the order specified in the query, after processing the first two predicates, a FOREST object that is contained in India is chosen. Upon executing the third predicate, which is the *Type* function, a greater constraint is imposed on this chosen FOREST object such that it should also be of type "evergreen" in order for it to satisfy this predicate and proceed with the testing of the subsequent predicates. The same logic holds for executing the next two functions, *Usage* and *Contains*.

In a more complicated query, we may need to distinguish the different groups of functions which are applied to the same object class, such that the respective results generated after processing each of these groups of functions on that object class will be used in a unique way to calculate some intermediate results for the computation of the final results of the user query. Below shows another example where aliases are needed to specify the distinct groups of predicates which could be imposed on the same object class. Each of these groups of predicates does not interfere with one another in the sense that predicates of one group will not impose a greater constraint on the same object class which is given distinct alias name by another group of predicates even though they are connected to one another via the logical AND connective.

Example 5-4. Find the lakes which intersect with an evergreen forest that is used for commercial purposes and a recreational forest that contains a waterfall in it.

```
F1 alias FOREST
F2 alias FOREST
select LAKE
where
    Type(F1, "evergreen")
And
    Usage(F1, "commercial")
And
    Usage(F2, "recreational")
And
    Contains(F2, WATERFALL)
And
    Intersects(LAKE, F1)
And
    Intersects(LAKE, F2)
```

In this example, the first two functions that can be imposed on the FOREST object class, with the alias F1 are *Type* and *Usage*. This F1 is not affected by the third and fourth functions, *Usage* and *Contains*, which are associated with the same original FOREST object class but with alias F2. Here, the first group of functions associated with F1 are imposed against all instances of the original FOREST object class. The same is true for the second group of functions associated with F2. Thus, the two groups of functions will eventually

generate two sets of FOREST objects. F1 will be associated with all evergreen forests of commercial usage in the original FOREST object class, and F2 will be associated with all recreational forests that contain waterfall. After that, if there exists a lake that intersects at least one instance from F1 and another instance from F2, then this LAKE object is selected as one of the final results to the query.

Objects generated as a result of processing a query are called *query-generated objects.* Here, we will discuss only the query-generated geographic objects. Any query-generated geometric objects produced are completely transparent to the user and thus, need not be specified by the user in writing an OFQL query. Below are some of these examples:

As has been discussed in Section 2.2.2, query-generated object class can be seen as selection on existing objects of an object class.

Example 5-5a. The example below generates a query-generated object class EVGRFOREST which instances are made up of a subset of objects from the object class FOREST which are of type "evergreen" and which contain at least a lake. Future reference to this particular object class of forests could simply done via the query-generated object class name EVGRFOREST.

```
object class EVGRFOREST:(FOREST)
select EVGRFOREST
where
      Type(FOREST, "evergreen")
   And
     Contains(FOREST, LAKE)
```

Query-generated object class could also be a relationship object class if it consists of several non-primitive component object classes.

Example 5-5b. The query-generated object class ADJRR is a new object class consisting of two different non-primitive object classes, namely: LAKE and ROAD. The following query produces tuple objects of the form <$LAKE_i$, $ROAD_i$> where each LAKE object, $lake_i$, is adjacent to the ROAD object, $road_i$ in each tuple object. Here, each tuple object is said to consist of components of LAKE and ROAD object classes respectively.

```
object class ADJRR:(LAKE, ROAD)
select ADJRR
where
     Adjacent(LAKE, ROAD)
```

If a query-generated object class consists of more than one object class, then to refer to the instances of one of the component object classes of this query-generated object class will need to explicitly specify both the relevant component object class name and the query-generated object class name connected with the **OF** keyword. In the example above, LAKE of ADJRR and ROAD of ADJRR are used to specify all those lakes that are adjacent to some road(s) and roads that are adjacent to some lake(s) respectively. LAKE of ADJRR and ROAD of ADJRR have the same object class definition as the original LAKE and ROAD object classes from which they are derived respectively.

Example 5-5c. Assuming we have a superfunction *Closest* which returns the closest object to a specified input object and the distance between the two, then the resulting query-generated object class WFD in this example, will consist of two non-primitive object classes, WELL and FOREST, together with the attribute function *Distance* which will be mapped to an INTEGER value that represents the distance between the two. After executing the query below, each WFD tuple object is a triple of WELL, FOREST and Distance where the FOREST object is the closest forest to the WELL object.

```
object class WFD:(WELL, FOREST, Distance)
select WFD
where
    Closest(WELL, FOREST, Distance)
```

A query-generated object class which consists of two or more non-primitive component object classes could not reference directly the functions of its component object classes. In order to reference them, we have to specify explicitly the relevant component object class of the query-generated object class.

Components of a query-generated object are always treated as one unit. Thus, a selection on those tuple objects of the query-generated object class WFD in the example above will always select all the three component objects as a whole unit.

Let us look at the left table of Figure 5-1 which shows all the instances of WFD. Suppose there is a query which wants to generate a query-generated object class WFD2 that consists of WELL, and FOREST pairs from the original WFD object class where the distance between a well and its closest forest is 30 meters as expressed in the query below, the resulting instances in WFD2 object class will be the same as the table shown on the right of Figure 5-1.

```
object class WFD2:(WELL, FOREST)
select WFD2
where
    Distance(WELL of WFD, FOREST of WFD, 30)
```

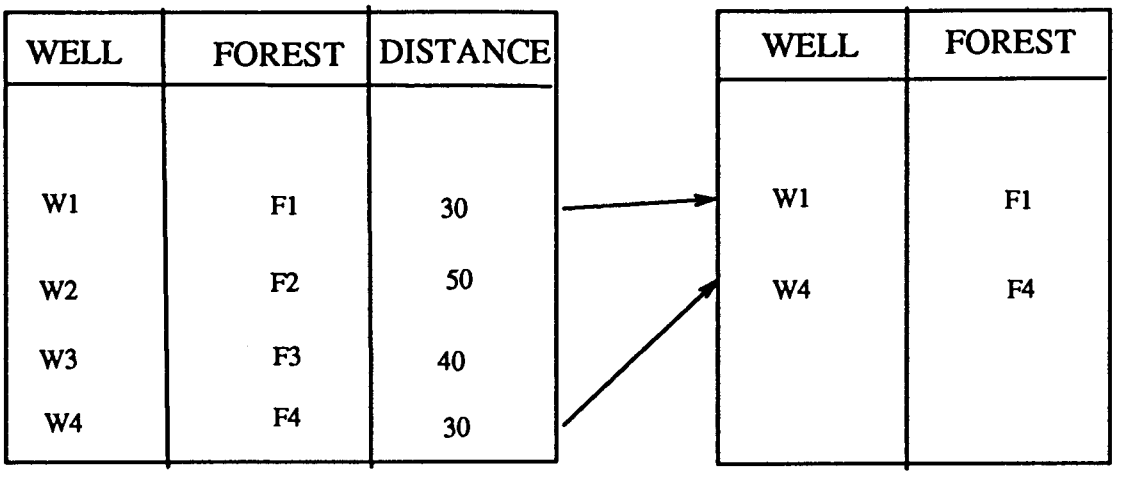


Figure 5-1: Query-Generated Object Class

Finally, query-generated object class could also consist of some newly-generated objects which are computed at run time. An example of this is given in Example 5-5d. Here, LAND is the newly-generated object class which consists of LAND objects that are situated in a commercial zone and have unstable soil.

Example 5-5d. Find the land in a commercial zone which soil is unstable.

```
select LAND
where
  Type(ZONE, "commercial")
And
  Type(SOIL, "unstable")
And
  Intersects(ZONE, SOIL, LAND)
```

Query-generated object classes are of utmost importance which model the new information obtained. Query-generated objects that satisfy all the criteria based on some properties or attributes, and relationships among instances from the same object class or different object classes may correspond to the new map layer that is produced as a result of overlaying different layers that satisfy some selection criteria in the traditional way of processing done by the existing GIS systems.

Let us take an actual application in urban growth and development for example. It is desirable for the

manager to determine the suitability of land for different usage purposes like agriculture or urban use and development. Thus, we may need to find all the LAND objects which are capable of supporting urban development by giving some selection criteria like those LAND objects that exclude marshes, steep slopes or LAND objects which are unsafe for the construction of buildings. These could be done with a sequence of *Complement* semantic spatial functions connected to each other with the logical AND connective. The LAND objects which satisfy all these conditions are generated using an OFQL query and a new query-generated object class LAND-FOR-URBAN-DEVELOPMENT is produced. User may store the LAND instances associated with this newly created query-generated object class permanently in the database for future use, or he could reference it in that user session where he does all the plannings and decision-makings and could be discarded automatically when he terminates that session without explicitly saving the query-generated objects. Using the OFQL user query interface, we can also determine the land area that are suitable for agricultural use which should not be converted for urban use.

5.2. Function Interface

To provide a user-friendly environment, the system usually provides an icon interface where the user could simply click on the mouse to invoke the desired operation. In our data model, property functions, superfunctions, and any other functions which have a specific object class as output could allow the user to click on the input object and the system will automatically display the output objects of the invoked function without any need to use the OFQL interface. For example, if we click on a particular COUNTRY object on the screen and click on its Population function in the pop-up menu, the numeric value which is mapped to by the Population function will be automatically returned. If we click on a particular parcel and click on the relationship function, OwnedBy, then the person who owns the parcel will be returned. Neighborhood is an example of superfunction which could be used at the clicking level without the need of a query language. For example, by clicking on a CITY object and choosing the Neighborhood function in the pop-up menu, the system displays or returns all its neighboring cities. Notice that superfunctions give an extensive set of custom-built functions that are made available to the end-user just like any other functions through the pop-up menus. There are some spatial functions which could not be implemented with just two clicks of mouse if either they have more than one input parameters or that they could have different choices of output object classes. An example is the *Contains* function, which output object class could be a well, forest, river and others. Here, additional number of clicks will be needed to retrieve the relevant information.

All the simple functions in our data model can be easily implemented to accommodate this clicking interface. For pure retrieval of straightforward information, user does not have to write query using OFQL but pointing devices such as the use of mouse interfaces and icons suffice.

Chapter 6
Query Processing

In this chapter, we assume the use of a single sequential processor in discussing some issues in processing OFQL queries where functions are all connected together via the logical AND connective. We further assume that given a single object as input parameter, a function will map it to *all* relevant objects where such a mapping exists. We will consider mainly the semantic spatial functions and retrieval functions in our discussion.

In our query processing scheme, a specific object of a certain object class is mapped to a set of objects of the same or different object class via a given function. Further processing could then proceed with each of the objects in this generated set. Notice that only the resulting objects being mapped to by a given function in a query can be considered for processing by the subsequent functions since they are the only ones where a mapping exists, and thus, satisfy the function. All other instances of a specified object class parameter where there exists no such mapping via a given function in the query, are not considered since they could not make the predicate true. Unlike [Kim, Kim and Dale 88], [Kim, Kim and Dale 89] and [Kim 90], this scheme does not need different ways for handling cyclic and acyclic query processing. Our approach allows cyclic query, acyclic query and single query consisting of a mixture of both cyclic and acyclic chains to be processed in the same way. The details will be given later on in this chapter.

Another aspect where our method differs from existing query processing techniques is that an OFQL query can be processed by groups of functions (or chains) aside from the conventional way of processing by individual functions (or edges). Processing a group of functions as a single unit allows the possible use of existence check with depth-first search technique where savings in both memory space and computation power could be achieved through prunings. Furthermore, depth-first search is a natural way of processing queries in an object-oriented environment that supports **clustering** where related objects are clustered together in the underlying storage structure and **object caching** where all related component objects are cached into memory when their aggregate parent object is retrieved. With object clustering and object caching, it is very likely that an object will be mapped to some objects that are *already cached in*, thus, making separate disk access for the retrieval of the relevant objects being mapped to after the processing of

each function in the chain unnecessary. This approach is totally different from the multiple JOINs of relational databases where processing is basically done by edges, after which a sequence of JOIN operations are performed to eventually generate the final results. It is also different from set processing of relational databases where big relations have to be accessed even if we are interested only in a few number of relevant objects. Finally, we also incorporate the use of the PART-OF and IS-A abstraction hierarchies for query processing aside from using them as data modeling constructs in our data model.

The generation of a query processing plan given any OFQL queries involves the use of the abstraction hierarchies of our geographic object data model presented previously in Section 4.1.2. After which, query is mapped into its corresponding QP graph, which is a general graphical representation of the query processing plan. PART-OF abstraction hierarchies of the geographic object data model, together with inverse functions and other heuristic rules are then used to identify the chains and edges in a QP graph. Proper chain processing strategies have to be determined for each chain that is selected for computation. Node collapsing, a process which will be described in the subsequent discussion, will then take place.

The rest of this chapter is organized as follows: QP graph is introduced in Section 6.1. Section 6.2 presents the query processing scheme. It includes a discussion of pre-processing, mapping an OFQL query into its corresponding QP graph, chain identification, chain processing strategy determination and node collapsing.

6.1. QP Graph

A query processing graph or the **QP graph** is the general graphical representation of the query processing plan generated by the query processor. It basically consists of a set of **unique labeled nodes**, each of which represents a distinct non-primitive object class, alias, component object class of a relationship object class, or a relationship object class that is used as parameter of a function in the query, a set of **unique labeled boxes** which shows the associativity of two or more non-primitive object classes that are not of the same existing relationship object class and are used together as either input or output parameters of a function and a set of **unique labeled directed edges** that connect the nodes and/or boxes together to represent the semantic spatial functions and/or retrieval functions that use the object classes represented by these nodes and boxes as their parameters. Note that an object class that is used as a parameter for several functions in a query will be represented by just one labeled node. Objects from different nodes enclosed in a labeled box are independent and do not relate to each other as tuple objects. Tuple objects of relationship object class, on the other hand, has their components related to each other in a specific way. Relationship object class is represented in the QP graph as a big node enclosing the nodes representing its non-primitive component object classes. From

now on, we will call this big node representing the relationship object class as the *enclosing node*. It is possible that two or more relationship object classes consist of exactly the same component non-primitive object classes except that the components of their tuple objects are related to each other via a different relationship. In this case, the component object classes of different relationship object classes are given distinct names by specifying the component object class name together with the keyword *of*, followed by the relationship object class name. This allows us to distinguish among the different relationship object classes even if their non-primitive component object classes are the same. For example, given two relationship object classes ADJFR and CNTFR. Each of them consists of two component object classes, FOREST and RIVER, such that tuple objects of the ADJFR relationship object class are FOREST-RIVER pairs which are spatially adjacent to one another while tuple objects of CNTFR are FOREST-RIVER pairs where the FOREST component of the tuple object spatially contains the RIVER component. Their QP graph representation is shown in Figure 6-1.

Since each node in the QP graph represents a non-primitive object class, it is only the semantic spatial functions and retrieval functions with one or more non-primitive object classes in both its input and output parameter lists that are being represented in the QP graph. We call this type of functions the **relationship functions**, whereas those semantic spatial functions and retrieval functions with either their input or output parameters that are solely of primitive object classes and are being mapped to or from one or more non-primitive object classes are called the **non-relationship functions** which are not represented in our QP graph.

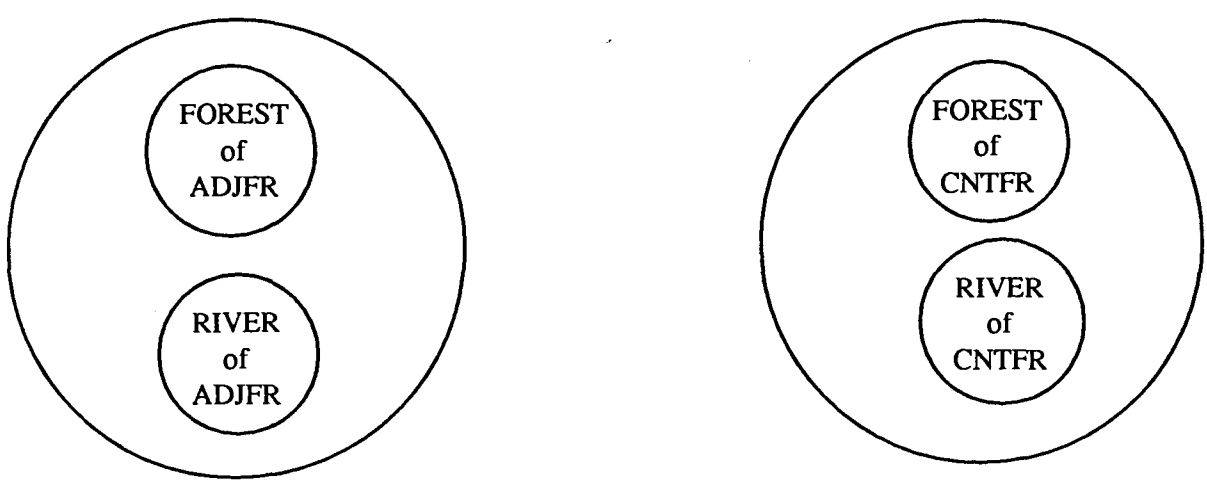


Figure 6-1: QP Graph of Relationship Object Class

Below are some examples by which a function is mapped into its corresponding QP graph.

Example 6-1. Please refer to Figure 6-2. Given the function f(a) -> b, the two unique labeled nodes *a* and *b* represent the distinct non-primitive object class parameters *a* and *b* of the function *f* respectively. The tail of the directed edge labeled *f* (which represents the function *f*) emanates from node *a* which is its input parameter and its head points toward node *b* which is the output parameter of the function. A practical example of such function in GIS application domain is Adjacent(FOREST) -> RIVER. Here, the *Adjacent* spatial function simply involves two non-primitive object classes, FOREST and RIVER. It generates all FOREST and RIVER object pairs that are spatially adjacent to one another.

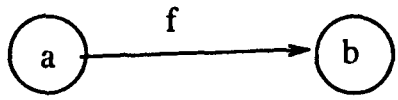


Figure 6-2: QP Graph for Example 6-1

Example 6-2. Please refer to Figure 6-3. Given the function f_3(a, b) -> c, a unique labeled box is drawn to enclose the nodes representing the non-primitive object classes *a* and *b* to show their associativity as inputs of the function f_3. Note that the box could be uniquely labeled by catenating the function name together with the name of all the object classes in its I/O parameter list. Here, nodes *a* and *b* are not in an enclosing node because *a* and *b* are not of the same existing relationship object class. An edge representing function f_3 emanates from this unique labeled box *f3abc* and points toward the single node *c* which is its only non-primitive output object class. An example of such kind of function is Route(C1, C2) -> PATH where PATH objects are the routes that connect the two CITY objects together, assuming that C1, and C2 are both aliases of the non-primitive object class CITY declared prior to the execution of this function.

As for the case where a function has a single non-primitive input object class parameter and two or more non-primitive output object class parameters, its QP graph representation will be a directed edge connecting the single node representing the only non-primitive input object class parameter to the labeled box enclosing the nodes representing the non-primitive output object class parameters of the function.

Example 6-3. Please refer to Figure 6-4. Given the function f_4(a, b) -> d where *a* and *b* are two of the non-primitive component object classes of a previously computed query-generated object class that consists of three non-primitive component object classes *a*, *b* and *c*. In this case, an enclosing node is used to

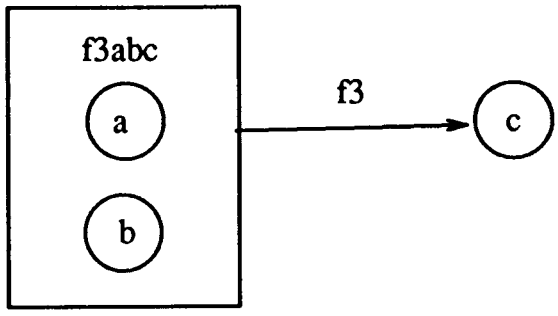


Figure 6-3: QP Graph for Example 6-2

represent this query-generated object class that encloses its component nodes *a* and *b* which are the only component object classes concerned for this query. The function f_4 is represented by the edge f_4 emanated from the enclosing node and points toward node *d* which represents its non-primitive output object class.

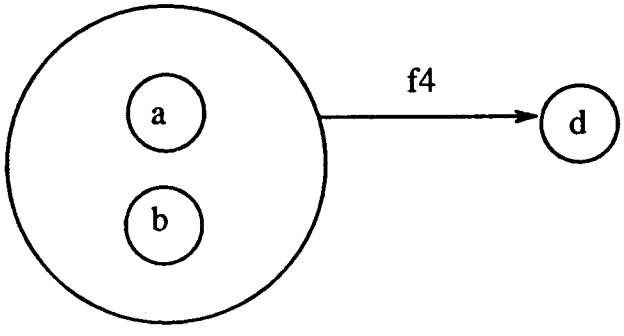


Figure 6-4: QP Graph for Example 6-3

The processing of the edge f_4 means checking if the a_i and b_i components in each tuple object $<a_i, b_i>$ of the query-generated object class has a mapping with an object d_j of the object class *d* via the function f_4. This is different from having a labeled box enclosing nodes *a* and *b*, because a labeled box merely tells us that *a* and *b* are used together as input or output parameters of a certain function but there implies no previous existence of $<a_i, b_i>$ tuple objects where a_i and b_i have a predetermined relationship with each other.

6.2. Query Processing Scheme

The sequence by which the user specifies predicates in an OFQL query does not dictate their processing sequence in a query nor does it statically specify which of the object classes are to be used as input and output parameters of the function. Our query processor will generate a query processing plan that specifies the actual details by which the query has to be processed by using some heuristic rules.

A correct query processing plan is said to be generated if each of the edges in the QP graph is processed once and only once such that the final instances found in the only nodes left after processing the entire QP graph are returned as results to the user query. Our **query processing algorithm** is shown below:

1. Pre-processing
2. Mapping the query to its graphical representation (i.e. a QP graph)
3. Decomposing the current QP graph into chains and/or edges and select one of them for computation
4. Determining chain processing strategy if a chain is selected for computation
5. Replacing the processed edge or chain referred to in step 4 by a new node, a process called *node collapsing*
6. Processing all set and aggregate functions if possible (i.e. all the individual values required by set and aggregate functions are available)
7. Repeat from step 3 if there are some more edges in the current QP graph

The remaining discussion will only focus on the first five steps. Pre-processing, mapping OFQL query into QP graph, chain identification, chain processing strategy determination and node collapsing are discussed in Section 6.2.1 through Section 6.2.5 respectively.

6.2.1. Pre-processing

As was mentioned at the beginning of the chapter, our discussion will mainly concentrate on the semantic spatial functions and retrieval functions. Set and aggregate functions which include the proximity functions and geographic spatial set functions like *Closest*, *Farthest* and others are all assumed to be performed only after their object class parameters represented as nodes in a QP graph have all their edges processed. This is

so done because of the way our query processor interprets queries. Here, set and aggregate functions are meaningful only when they are performed on the final relevant set of objects which satisfy all the other functions in the query.

All those non-relationship functions should be processed first prior to the construction of the QP graph. An example of this is the function f_2^{-1}(a) -> b where *a* is a primitive object class and *b* is a non-primitive object class. In this case, function f_2^{-1} has to be processed first, after which only a single node representing the non-primitive object class *b* which contains only all the b_i instances that has a mapping with the primitive object class *a* via the function f_2^{-1} is shown in the QP graph. A practical example of such function is the predicate *Type(FOREST, "evergreen")* which upon execution, will select only all the FOREST objects which are of evergreen type. After that, its corresponding QP graph representation can be constructed, which is only a single node representing the non-primitive object class FOREST. Here, it implies that all functions which can instantiate their object class parameters with constants should definitely be processed first prior to the construction of the QP graph. This allows the immediate restriction of search space on the relevant objects. Eventually, what will be portrayed in the QP graph are the mappings among non-primitive object classes via the relationship functions. This strategy is analogous to the *select before join* query processing heuristic in relational database.

If we have a function f_3(a, b) -> c where *a* and *b* are non-primitive object class parameters of the function f_3 and *c* is a primitive object class, then the input parameters are represented as an enclosing node with two component nodes *a* and *b* in the QP graph. Labeled box is not used for this case because the function f_3 is *processed* prior to the construction of the QP graph, such that a_i, b_i in the generated tuple objects <a_i, b_i> of the enclosing node are related to one another via f_3. Note that *c* is not represented in the QP graph because it is a primitive object class.

However, non-relationship functions which involve query-generated object class parameters that are not previously existing should be computed only after processing the function or group of functions which generates its actual objects. An example of this is the function *Length(PATH)* -> *INTEGER*. Here, since the PATH objects do not exist prior to the processing of this function and that INTEGER is not of finite domain, so there is no way by which PATH can be used as an input or output object class parameter of this function. Thus, the processing of the non-relationship function should be delayed until all edges connected to the node representing the PATH object class in the QP graph are processed. This is the only exception by which non-relationship functions are not processed prior to the QP graph construction.

Before we proceed, let us define some terms which will be used in the subsequent discussion. **Target nodes** are the nodes representing the object classes specified after the SELECT clause of an OFQL query. Objects belonging to these **target object classes** that satisfy all the functions specified after the WHERE clause of an OFQL query are returned as final results to the query.

A **chain** is a sequence of nodes or boxes connected together by some directed edges that have the same orientation. Its **path length** is the number of edges in the chain.

There are two kinds of chains, namely: simple chain and cycle. A **simple chain** is a kind of chain where each node and edge in the chain could only be traversed once. Please refer to Figure 6-5 below for an example of a simple chain.

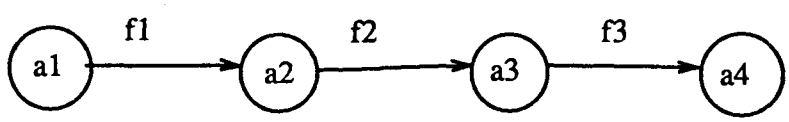


Figure 6-5: Example of a Simple Chain

A **cycle**, on the other hand, is a special kind of chain where each node and edge are traversed only once, except for one of the nodes, which is traversed twice. That particular node acts as both the starting and the ending node of the cycle. A cycle could be represented as follows: a_1 -> a_2 -> ... -> a_n -> a_1 with a_1 acting as the starting and ending node of the chain.

6.2.2. Mapping OFQL Query into QP Graph

Before the construction of the QP graph, each function in the user query which involves object classes that are at the non-leaf nodes of an IS-A abstraction hierarchy needs to be mapped first into several functions. The number of functions generated is equal to the number of leaf nodes in that IS-A class hierarchy concerned. Each of these functions in turn substitutes the originally given non-leaf object class parameter by an object class that is at the leaf nodes of that IS-A class hierarchy. Recall in Section 4.1.2 that a non-leaf object class of an IS-A class hierarchy is a non-spatially-associated geographic object class. It could be implemented as a directory which points to its subclasses. All object classes located at the non-leaf nodes of an IS-A class hierarchy are but just empty containers without any object instances. Only object classes at the leaf nodes have the actual object instances directly associated with them. Thus, if a function has an object

class parameter which is at a non-leaf node of an IS-A class hierarchy, that IS-A class hierarchy will need to be traversed first until the leaf nodes which represent the child object classes are reached.

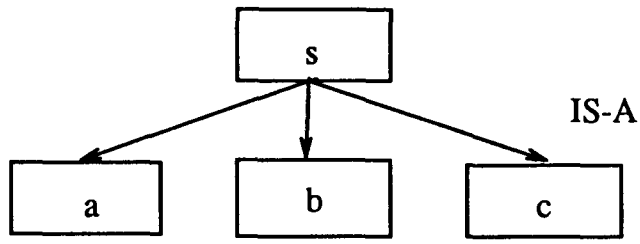


Figure 6-6: IS-A Class Hierarchy

Please refer to Figure 6-6. Consider an example where *f(s)* -> *e* is a function with a superclass *s* of a particular IS-A class hierarchy, and another object class *e* as its I/O parameters respectively, and that *a*, *b* and *c* are the object classes at the leaf nodes of the IS-A class hierarchy associated with *s*, then this function *f(s)* -> *e* will be mapped into three separate functions, *f(a)* -> *e*, *f(b)* -> *e*, and *f(c)* -> *e*, using *a*, *b* and *c* as their parameters respectively instead of using the superclass *s*.

After all functions in a user query involving non-leaf object class parameters of an IS-A class hierarchy are mapped into their corresponding functions with leaf node object class as parameters, the query processor will then process all the *Contains* functions in a user query which parameters are of the same PART-OF hierarchy.

For example, given the PART-OF hierarchy in Figure 6-7, *Contains(COUNTRY)* -> *DISTRICT* is one of the *Contains* functions in a user query. This function will then be mapped into the following functions: *Contains(COUNTRY)* -> *PROVINCE*, *Contains(PROVINCE)* -> *CITY*, and *Contains(CITY)* -> *DISTRICT*. As can be seen, for each *Contains(a)* -> *b* function in a query, it will be substituted by *n* functions where *n* is the path length or distance between its two object class parameters *a* and *b* in the PART-OF hierarchy. Each of these *n* functions is a *Contains* function between an aggregate parent object class and its immediate component object class in that PART-OF hierarchy and that *a* and *b* are parameters in two of these *n* *Contains* functions respectively. All these *n* *Contains* functions are later on assembled as a single chain as will be discussed in the next section.

Now that we have the complete set of functions with all the IS-A and PART-OF hierarchies resolved, and that all the non-relationship functions are processed, the query processor can then proceed with the

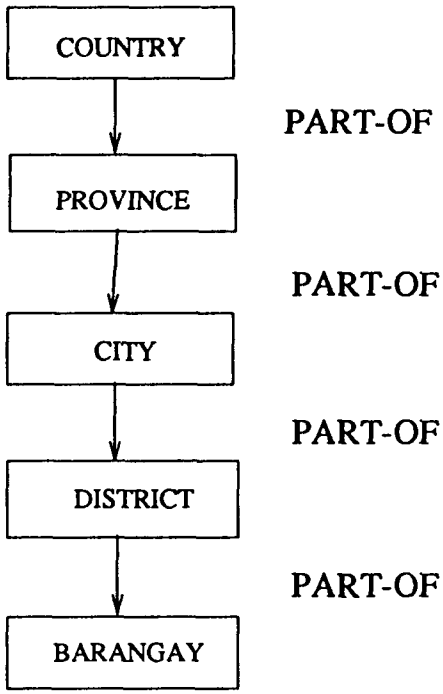


Figure 6-7: PART-OF Hierarchy

construction of the QP graph for all the remaining relationship functions in the query. Details of QP graph construction are given in Section 6.1. As we may have noticed, the actual I/O parameters of the functions are not yet determined as of this stage, thus, the orientation of the edges and the possible use of labeled boxes in the QP graph are still unknown. We will present in the next section the use of PART-OF abstraction hierarchies, inverse functions and other heuristic rules in the dynamic identification of chains and edges as I/O parameters of functions in the query are determined.

6.2.3. Chain Identification

The key to chain identification is in the determination of proper edge orientation. First of all, query processor has to identify those edges which result of execution could be directly derived from the upward containment transitivity of the same PART-OF hierarchy. Nodes which are connected to this type of edges and are found in the same PART-OF hierarchy must be grouped together as a single simple chain, so that containment relationship can be directly derived without the need of performing actual spatial or geometric computations nor consultation of spatial indices. The *Contains* functions to which each of the *Contains* functions in the original user query are mapped to, as is discussed in the previous section, constitute a chain.

The advantage of the PART-OF abstraction hierarchy in query processing is in its direct access of the aggregate parent objects and/or component objects given any objects that participate in the PART-OF hierarchy.

There are several implementations of a PART-OF hierarchy. The aggregate parent object may use a labeled pointer to link to its component objects, *or* vice versa. In this case, the pointers are uni-directional, that is either pointing in a top-down or bottom-up direction of the PART-OF hierarchy. Another scheme is using bi-directional labeled pointers where each object class has two sets of pointers. One set points to its aggregate parent object(s) and the other set points to its component objects. The latter scheme offers much flexibility since given any instantiated objects that participate in the PART-OF hierarchy, the query processor can always retrieve their query-relevant aggregate parent or ancestor objects and component objects directly.

If the PART-OF abstraction uses uni-directional pointer scheme, two cases may occur:

- *Case 1:* The pointers' direction is from child to parent.

  Bottom-up strategy is used here where many irrelevant objects are considered at the very beginning since all the instances in the object class represented as leaf node in the PART-OF hierarchy are being considered even if the root of the hierarchy is instantiated in the query. This is due to the fact that there are no pointers from the instantiated aggregate parent object to its component objects. This scheme is good for modeling $Contains^{-1}$ function with the leaf node in the PART-OF hierarchy instantiated and is asking for the enclosing object (or the aggregate parent object). Using Figure 6-7 as an example, the former function *Contains(COUNTRY) -> DISTRICT* in the user query will be processed in the following sequence: $Contains^{-1}(DISTRICT)$ -> *CITY*, $Contains^{-1}(CITY)$ -> *PROVINCE*, and $Contains^{-1}(PROVINCE)$ -> *COUNTRY*. Note that here, the inverse function, $Contains^{-1}$, must be used because only *uni-directional* pointers from child to parent are available.

- *Case 2:* The pointers' direction is from parent to child.

  In this case, we can start immediately with the relevant instance if the aggregate parent object in the PART-OF hierarchy is instantiated. This scheme is good to model the *Contains* function given an instantiated aggregate parent object. However, if it is the leaf node in the PART-OF hierarchy that is instantiated and $Contains^{-1}$ function is invoked to find its aggregate parent object, then using this scheme will require all aggregate parent objects to be considered. Any

objects in the leaf node level which are mapped to by the aggregate parent objects using the PART-OF hierarchy that match any of the instantiations specified for that leaf node by some other functions in the query will have their corresponding aggregate parent object from which they are mapped, returned as result to the *Contains*$^{-1}$ function. Note that the total number of instances in the aggregate parent object class that have to be considered is always less than the number of instances in the leaf node object class, thus, the total number of objects considered using this scheme is smaller than that using the scheme presented in case 1 above. Using Figure 6-7 again as an example, the function, *Contains(COUNTRY) -> DISTRICT* in the original user query will be processed in the following sequence: *Contains(COUNTRY) -> PROVINCE*, *Contains(PROVINCE) -> CITY*, and *Contains(CITY) -> DISTRICT*.

For the case where uni-directional labeled pointers are used, orientation of edges in a simple chain representing the containment relationship among object classes of a particular PART-OF hierarchy becomes known depending on whether the pointers' direction is from child to parent or vice versa. In our discussion, we will assume the use of bi-directional labeled pointers where either forward or reverse traversal scheme can be used. These two traversal methods determine the edge orientation of the PART-OF or *Contains* chain, that is, to decide whether the search should start with the aggregate parent object or the component objects of the PART-OF hierarchy.

Let us now consider the **forward traversal** scheme which determines first the enclosing object to limit the search scope of subsequent functions on objects that are within a specific geographic area. This scheme can limit the search space of a query in its earliest stage of processing. It is particularly attractive when objects satisfying other spatial and non-spatial functions are scattered over a wide geographic area. If the query processor performs these other spatial and non-spatial functions first, it may possibly be considering a lot of irrelevant objects. If it makes use of the PART-OF hierarchy at the very beginning in processing this user query, then the search scope can be restricted to a particular region on the earth's surface. The primary goal of query optimization is to be able to generate results to a query by processing only the relevant objects and discarding all irrelevant objects from consideration at the earliest stage possible such that the number of times by which each function routine needs to be executed will be minimized. The PART-OF abstraction hierarchy does contribute to this objective by early restriction on the geographic area that has to be considered.

Let us illustrate our discussion above with an example.

Example 6-4. Find the cities of India which have a forest that is adjacent to a river.

The OFQL query could be written as follows:

```
select CITY
where
    Name(COUNTRY,"India")
And
    Contains(COUNTRY, CITY)
And
    Contains(CITY, FOREST)
And
    Adjacent(FOREST, RIVER)
```

The first function is a non-relationship function and should be processed first prior to the construction of the QP graph for this query. Since it is instantiated to only one geographic object, the execution of this function can immediately limit our search within India. Suppose the containment relationship between COUNTRY and CITY is specified in the PART-OF hierarchy, the Contains(COUNTRY, CITY) function needs to be performed next before the last two functions are executed since the containment relationship can be directly derived without doing any spatial computations, in which case, only forests from the cities of India are checked for adjacency with rivers. If forward traversal scheme is not used, the query processor will then have to check all the cities in the world that have forests. In the latter case, functions are possibly invoked for more number of times than is necessary since a lot of irrelevant cities and forests are considered in the computation.

Reverse traversal, on the other hand, starts with all the relevant objects that satisfy all other functions in the query before the *Contains* function is executed to determine exactly the target instances that could be returned as results to the user query. This scheme is attractive if the object class parameters of the *Contains* function are not in the PART-OF hierarchy, and the number of instances that satisfy all other non-spatial functions is small. In this way, only very few objects need to be considered in the actual spatial containment computation.

After the selection of forward and reverse traversal, we will know whether the chain representing the group of *Contains* functions should be placed at the beginning or at the end of a longer chain that is to be assembled. The determination of the orientation of the other edges of this longer chain is presented in the discussion below.

For each function, there is a fixed number of object classes that could be specified as its inputs and outputs respectively. There are also some constraints as to which object classes could be used as its I/O parameters.

The query processor has to take all these into consideration and specify the orientation of edges where inverse functions are not applicable.

For this type of function, the individual object class parameters are statically classified as either input(s) or output(s) of the function. A simple example of this is the function Route(C1, C2) -> PATH. Here, the PATH objects are query-generated objects. They do not exist prior to the execution of this function. Thus, its inverse function Route^{-1}(PATH) -> (C1, C2) is meaningless since PATH is undefined. As can be seen here, those query-generated objects which do not exist prior to the execution of the function that produces them, cannot be used as input object parameters of any functions. However, upon the generation of the query-generated object class after the execution of the function which produces its actual objects, that query-generated object class can then be used as an input parameter of the subsequent unprocessed functions which have it as one of the I/O parameters.

In the above example of the *Route* function where PATH objects do not exist prior to its execution, its inverse function *Route*$^{-1}$ could not be used. Hence, the orientation of the edge representing this function is fixed such that the tail of the edge emanates from the labeled box enclosing the two labeled nodes *C1*, and *C2*, and points toward the node PATH which is the output parameter of the function.

At this stage, after the identification of all those edges in a QP graph which orientation is fixed, the remaining ones represent functions which have inverse functions. In this case, the orientation of these edges in the QP graph becomes arbitrary and may evolve depending on whether the use of the original function given in the query or its inverse function can offer a better way of processing.

Let us consider Figure 6-8 and Figure 6-9 for examples of graph transformations into uni-directional chain and uni-directional cycle by using inverse functions respectively. The graph before the application of inverse functions and the graph after the application of inverse functions mean the same thing, and will eventually generate the same results to the query.

By the definition of inverse function discussed in Section 4.1.3, we say that if the mapping from an object class to some object classes via a given function f is generated, then the mapping using the inverse function f^{-1} could be directly obtained from the mapping generated by function f and vice versa. Hence, the query processor could choose either the original function or its inverse in processing each function (where inverse function is applicable).

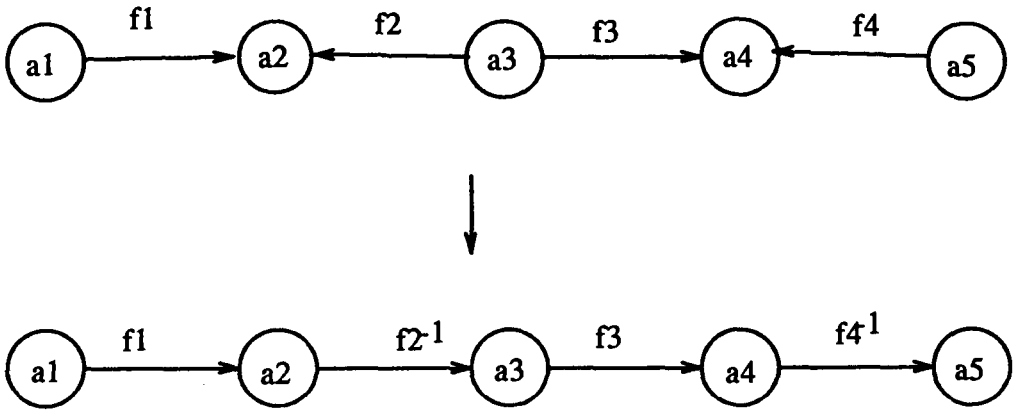


Figure 6-8: Chain Using Inverse Functions

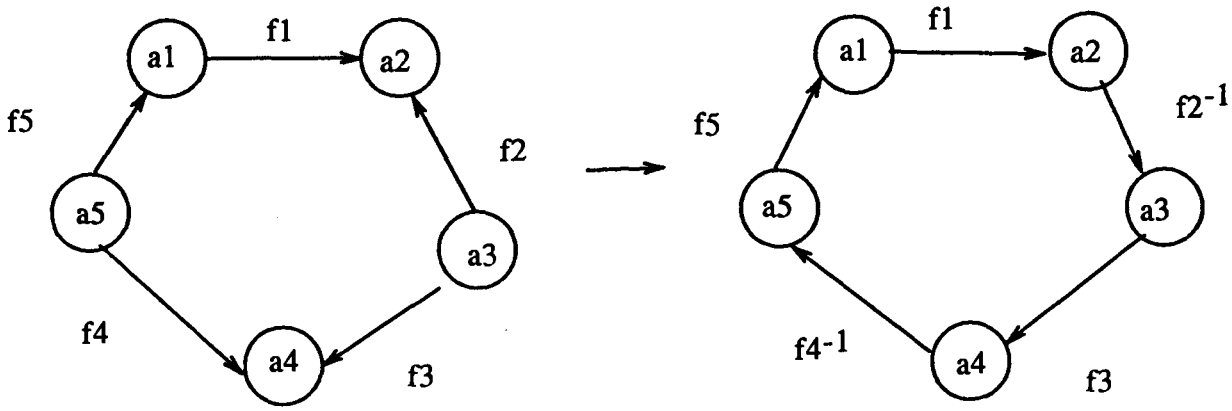


Figure 6-9: Cycle Using Inverse Functions

The major motivations of chain processing are the reduction in space and computation as a direct consequence of the use of existence check and depth-first search which will be discussed in Section 6.2.4.2, and that chain processing is very appealing when used in an object-oriented environment where clustering of relevant objects and object caching are supported. In this case, there is a greater probability that the relevant objects which are to be mapped to by the functions (processed in a depth-first sequence) are already in the cache. Thus, it is desirable for the query processor to assemble edges into long chains. The optimal path length of a chain that is to be assembled depends on the object caching and object clustering schemes of the involved object classes represented as nodes in a chain.

The simple chain identified as a result of using the PART-OF hierarchies and the edges which have fixed orientation could be assembled together with the remaining edges which orientation is arbitrary (where inverse functions are applicable) into a long chain. If this is not possible, the result will be a set of chains and edges. Suppose node *a* in Figure 6-10 has two edges connected to it such that these two edges are of different orientation, then node *a* is said to participate in two separate chains, namely, f_1-f_2-f_3 and f_4-f_5.

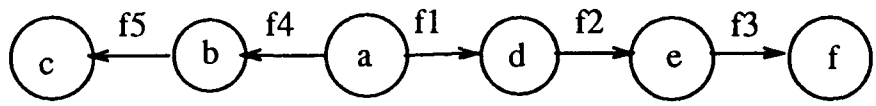


Figure 6-10: Node *a* Participating in Two Chains

Some other heuristic rules that can be used in assembling long chains include the processing of non-spatial functions prior to spatial function execution, and the processing of functions which results can be derived using indices prior to the execution of those which need actual computations or do not have indices. The motive for the first heuristic rule is to restrict the number of objects which need to invoke spatial functions where mapping to geometric representations and invocations of geometric routines are required based on our bi-level data model. The rationale for the second heuristic rule is straightforward, and that is to take advantage of mechanisms which allow direct access of objects.

If a node has more than one outgoing edge, then the outgoing edge which can be mapped to the least number of output instances and which could be processed using indices should be processed first. This can limit the search space at an earlier stage of processing a query.

Although we have mentioned about the advantage of processing long chains in our model, however, we should be aware also of the fact that chain processing involving many interface nodes is very expensive which details will be discussed in Section 6.2.4.2 and 6.2.5.2. **Interface nodes** refer to those nodes in the current chain that are also connected to some other edges in the QP graph. In Figure 6-11 , nodes *b*, *c* and *d* are interface nodes of the current chain f_1-f_2-f_3-f_4-f_5. In other words, query processor has to weigh the pros and cons of assembling long chains considering the benefit of using depth-first search without ignoring the issue of interface nodes in a chain to eventually come up with the best decomposition for a given QP graph.

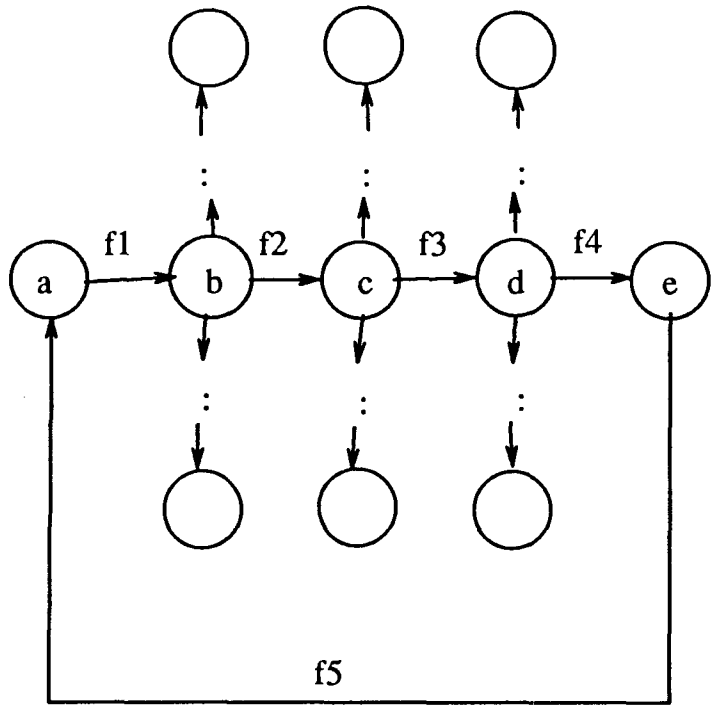


Figure 6-11: Long Chain with Interface Nodes

6.2.4. Chain Processing Strategy Determination

Chain processing evolves around two main issues, that is, the selection of root node and the selection of appropriate searching techniques. These will be discussed in Section 6.2.4.1 and 6.2.4.2 respectively.

Given a chain, one can envisage trees of objects which are called **search trees**. Each search tree has *n* layers, with edges connecting objects in one layer to the next (lower) layer representing the mapping defined by the functions. The number of search trees generated is equal to the number of instances in the starting node of a chain, which we call as the **root node**. An instance under consideration which belongs to the target object class is called a **target instance** and an object from the root object class which is used in the generation of a search tree is called a **root instance**. Note that with the definition of a search tree, it is actually an expanded version of a chain where each node representing an object class in the original chain is now representing the actual relevant *instances* in that object class involved in the actual mapping of functions

of a given query. Thus, all nodes in the same layer of a search tree, which we call as **instance nodes**, represent instances of the same object class. The number of layers in the search tree is equal to the path length of the chain which is also equal to the number of functions in the chain.

The discussion in the next two sections will be about the proper selection of root node, followed by an analysis of the searching techniques for processing chains.

6.2.4.1. Root Node Selection

Usually, the target node is used as the root node in the generation of search trees. If the target node is not the root node, then many branches generated may not really contribute to the detection of a *new* target instance that could be returned as a result of processing that chain since many target instances that are being mapped to may be previously generated by other instance nodes at the upper layers of the search tree. Whereas if the target node is used as the root node in the generation of search trees, each target instance is examined only once.

An exception to the above heuristic rule is if the size of the target node is too big compared to the size of other nodes in the chain, then the node with the smallest size is usually chosen as the root node in the generation of search trees. The size of a node refers to the number of instances in the object class represented by that node. The rationale behind this is that the number of search trees generated in a chain is equal to the number of root instances. The number of functions invoked by the lower-layer instance nodes is a multiple of the number of upper-layer instance nodes generated in a search tree. Thus, if the query processor starts out with a root node which has a very large number of instances compared to other nodes in the chain, then many objects are processed which may eventually find out to be irrelevant. If the query processor starts off with a small root node, then at least, the total number of search trees generated or the total number of times by which the functions are invoked and executed may be smaller assuming that the number of objects which invoke the functions is directly proportional to the number of output objects generated at its next lower-layer in a search tree. With the above assumption, the total number of function invocations and executions using this heuristic rule in the processing of the entire chain could be reduced.

6.2.4.2. Searching Techniques

There are two types of checking mechanisms which are used in chain processing to generate all the relevant target instances that can be returned as final results of processing that chain. These two checking mechanisms are *existence check* and *closure check*.

Existence check is used if there is only one target node in the chain and that target node is chosen to be the root node in the generation of search trees. It can benefit from depth-first search which fits well into object clustering and object caching. The primary advantage of using existence check is that pruning can eliminate a lot of unnecessary computations once a target instance is found to satisfy all the functions in the entire chain.

Closure check, on the other hand, is used when there are more than one target node in the chain or when the target node is not the root node. In this case, the portion of a chain starting from the root node up to the lowest level where the last target node or last interface node (whichever is at a lower level) in the chain is situated must use closure check to ensure that all relevant target instances which are possible candidates to be returned as results of the chain will be examined.

There are two ways by which closure check can be implemented. They are the *depth-first search* and the *breadth-first search* techniques. The depth-first search approach for closure check saves space storage but requires more number of function invocations due to the interface between the function implementor and the physical database in the architecture of our model. Breadth-first search, on the other hand, generates all relevant objects of each layer of the search tree with just one function invocation. This gives the advantage of less number of function invocations, but the space for storing the intermediate instance nodes could increase exponentially as each layer of the search tree is generated assuming that the number of output objects are directly proportional to the number of input objects used in the mapping.

First, let us discuss about existence check. When chain processing starts, existence check ensures that each of the instances in the target object class is checked and no repetitive examination will occur on the same target instance. We need only *existence check* for all other object classes in the other layers of the search tree to see if there *exists* a mapping originating from a target instance that can make the entire chain true. In other words, if there are j functions in the chain, and there *exists* a path of length j originating from a given target instance of the search tree, then that target instance is returned as one of the results of the chain, and pruning could be done on that branch. Processing will then proceed with the next root instance (which is also a target instance in this case).

The major motivation for using existence check is in pruning. Pruning discards all other intermediate objects generated along the branch which is originated from the selected target instance in a search tree. In this case, the number of function invocations and executions is reduced. It saves a lot of computations and reduces the number of intermediate objects generated at each layer of the search tree because not all nodes in

the branch are examined. Note that prunings may be significant since the number of times by which a function is invoked at each lower-layer in a search tree is a multiple of the number of objects that need to be examined at the upper layer. Thus, proper sequencing of functions in a chain is crucial to take advantage of possible prunings.

Prunings in existence check is particularly beneficial and more attractive than closure check if functions in a chain require complex computations. However, if the target node is not the root node in the generation of search trees, then it is better to put target node close to the root node in a chain, so that more edges in the chain can be processed using existence check. If there are more than one target node in the same chain, it is not enough to place only one target node close to the root node, since it is that portion of a chain starting from the level where the *last* target node or *last* interface node (whichever is at a lower-level of the chain) is up to the last level of the chain that can use existence check. Thus, all target nodes and interface nodes (that are connected to some other unprocessed chains or edges in the QP graph) should be placed near the root node in a chain.

Existence check can be implemented using the depth-first search technique. To give a fuller description of how depth-first search is carried out, we have the following discussion.

Each object in the upper layer of the search tree is mapped to a set of objects in the next lower layer which must be stored. We call the temporary storage for this set of mapped objects as the *intermediate table*, and each object in the intermediate table is called an *intermediate object*. Again one instance from this intermediate table is applied to the next function. If there is a branch in the search tree where there exists a mapping of objects from each consecutive layer of the search tree, such that all functions in the chain are satisfied, then the target instance is selected as one of the results to the given chain, otherwise, the traversal will backtrack one layer up the search tree to retrieve the next object in the intermediate table generated at that layer and process downward again. If an intermediate table at the lower-layer is exhausted and still no mapping exists, traversal will again go one more layer up the search tree. If eventually, the root of the tree is reached and no such mapping exists, then that target instance is not selected. Processing starts anew with the next instance in the target object class. If ever a mapping exists for that branch emanating from a particular target instance, then pruning occurs such that all the intermediate tables generated at each layer of the branch in the search tree are abandoned and that target instance is selected. The entire process repeats all over again with the next instance in the target root node. This entire process is summarized in Algorithm 1 for depth-first search technique.

Algorithm 1: Depth-First Search Technique

The algorithm consists of the main program called *Program_Start* and a recursive function called *Mapping* which checks if the instance nodes at layer *i* could be mapped to some instance nodes at $(i+1)^{th}$ layer. The *sizeof()* function returns the total number of elements in a set.

a is given as the target node and the root node in the generation of the search trees. There are *j* functions in the chain, thus, the path of the chain is of length *j*.

The variables used in Program_Start are as follows:

- *i1* is the index variable used to control iteration over all instances of *a*.
- *n1* is the total number of instances in node *a* (which signifies the number of instances in object class *a*).
- *nextlevelnode* is the output object class of the function or the next lower level node in the search tree to which instance nodes of the current input object class are to be mapped to. It is the actual parameter to the recursive function *Mapping*.
- *mapped* is a switch which is turned to TRUE by the recursive function, *Mapping*, if there exists a path of length *j* that emanates from a target instance a_{i1}. In this case, a_{i1} will be selected as one of the outputs to the chain.

The variables used in the recursive function, *Mapping*, are as follows:

- *i2* is the index variable used to control iteration over all instances of an input node or what we call as the intermediate table in the previous discussion for those which are used as inputs.
- *n2* is the total number of input instances to be considered in that function invocation.
- *inputnode* is the formal parameter of the recursive function, *Mapping*, which represents the input object class parameter of the function considered.
- *function#* is the index variable that determines the function to be applied in each invocation to generate the next layer of the search tree.
- $f_{function\#}$ represents the $function\#^{th}$ function in the chain.

- $inputnode_{i2}$ is the $i2^{th}$ instance node of *inputnode*.
- $outputnode_{function\#}$ represents the output node or the output object class to be mapped to by the $function\#^{th}$ function in the chain given an input instance node, $inputnode_{i2}$, at that invocation.
- $nextlevelnode_k$ is the output node or the output object class in the next invocation of the function, *Mapping*.
- *pathlength* is the number of functions in the chain.
- *mapped* is similar to the definition found in Program_Start which is used as a logical switch.
- *k* is the index variable of the next function to be used for mapping.

```
Program_Start()
begin
 for i1 = 1 to n1
     mapped = FALSE
     Mapping (a, 1, nextlevelnode_1, 1, j, mapped)
     if mapped then output a_i1
 endfor
end

Mapping (inputnode, n2, outputnode, function#, pathlength, mapped)
begin
 i2 = 0
 while i2 < n2 And mapped = False
    i2 = i2+1
    if there is a mapping such that
       f_function#(inputnode_i2) -> outputnode_function#
    then
       begin
          if function# = pathlength
          then
               begin
                  mapped = TRUE
                  return (mapped)
               end
          else /* function# < pathlength */
              begin
                 k = function# + 1
                 Mapping (outputnode_function#,
                       sizeof(outputnode_function#),
                       nextlevelnode_k, k,
                       pathlength, mapped)
              end
       end
 endwhile
 return (mapped)
end
```

Note that pruning takes place with the use of the conditions specified under the **while** statement of the recursive function *Mapping*.

Example 6-5: Let us use an example to illustrate the processing using existence check with depth-first search strategy. Consider the processing of the chain shown in Figure 6-12 where a is the target node. Using Algorithm 1, the search tree generated by a single target instance a_1 is shown in Figure 6-13.

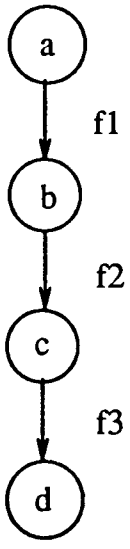


Figure 6-12: Chain for Example 6-5

Here, a single target instance a_1 is mapped to a set of instances b_1 to b_x via the function f_1. After that, using depth-first search, b_1 is taken as input to the next function f_2 to map to the relevant instances c_1 to c_n at the next layer. c_1 is then used as input to the function f_3, but could not find any mapping with the d instances. Thus, the next intermediate instance node c_2 from the second layer of the search tree (where the root is at the 0^{th} layer of the search tree), is then used as input to function f_3. This time, it is mapped to instances d_1 to d_z at the lowest layer of the search tree. Since there are only three functions in the original chain of Figure 6-12, so the height of the search tree is 3. Now that there exists a path of length 3 emanated from the target instance a_1, so a_1 is immediately selected as one of the results to this chain since a_1 satisfies mappings of all functions in the query. Note that here, c_3 to c_n, and b_2 to b_x are not processed and could be pruned immediately since only the target instance is of interest. Once a target instance is selected as one of the results, the rest of the intermediate instance nodes generated in that branch are pruned, thus, saving a lot of

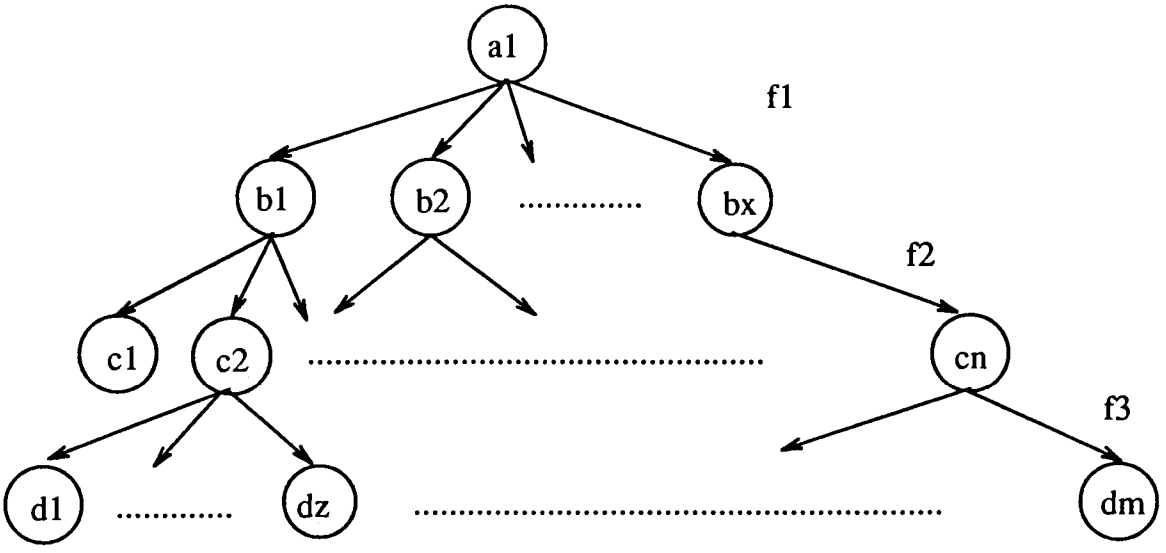


Figure 6-13: Depth-First Search Tree for Example 6-5

computations and space. This process of search tree generation will repeat using the next instance from the target node *a* as the root instance of the search tree. The entire process ends after *all* instances of the target root node *a* are examined.

Now, we are going to discuss about closure check. As we have mentioned in the early part of this section, closure check has to be used if the target node is not the root node in the generation of search trees. It is used to process the chain starting from the root node up to the level where the target node is situated (assuming target node is at a level lower than the last interface node in the chain). This ensures that all relevant target instances are examined and that all target instances that satisfy the chain will definitely be detected. After performing the closure check on the upper portion of this chain, the level where the target node is situated and the nodes and edges below it in the chain can simply use existence check for processing. In that case, the number of search trees generated for existence check is equal to the number of target instances generated in closure check. If a target instance in this selected set has a path which shows the existence of a mapping with the remaining layers of the search tree, it will be returned immediately as one of the results of the entire chain.

In the discussion above, we assume that after performing closure check on the upper portion of the chain, the remaining subchain that uses existence check uses the target instances generated in the closure check as the root instances. This is true if the **fanout** of the search tree is less than a given **threshold** such that the general structure of the search tree after mapping all the functions in a chain is skinny.

Fanout above refers to the number of objects being mapped to by all the instance nodes at a specific layer of the search tree via a given function in the query. The *threshold* determines the maximum number of instance nodes that could be held at each layer of the search tree. If the threshold is exceeded, then the search tree is said to be *bushy* and the chain may need to be decomposed into subchains and/or edges, otherwise, the structure of the search tree is said to be *skinny* if all layers in the tree do not exceed the threshold.

If the search tree generated using closure check for processing the upper portion of a chain is bushy, then the lower portion of that chain could be seen as a separate subchain where the target node includes only those relevant target instances generated by the closure check, and the root node in the processing of this subchain could be chosen as was discussed in Section 6.2.4.1. Processing will then proceed with this subchain.

For the case where a chain does not contain the target node, but just the interface nodes, then closure check is used for processing the portion of the chain starting from the root node up to the lowest level where the last interface node is situated.

There are two ways by which closure check can be implemented. They are the depth-first search and the breadth-first search. We will briefly discuss the difference and the pros and cons of each of these approaches.

Please refer to Figure 6-15 for the use of depth-first search in closure check in processing the chain shown in Figure 6-14.

Assuming *d* as the target node, the numbers beside each node indicate the sequence by which each node is examined. As can be seen, initially one object is used as input of a function which maps to a set of relevant objects in the next layer, afterwards, only one object from this newly generated set is then used as input to the next function. The traversal technique is depth-first.

Please refer to Figure 6-16 for the sequence by which each node is visited using breadth-first search technique. Here, all objects at the same layer are mapped to objects in the next layer at the same time. It differs from depth-first search technique in that instead of examining *only one* instance node and generating

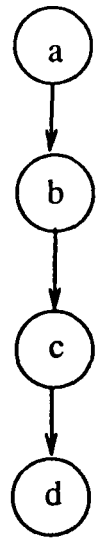


Figure 6-14: Chain

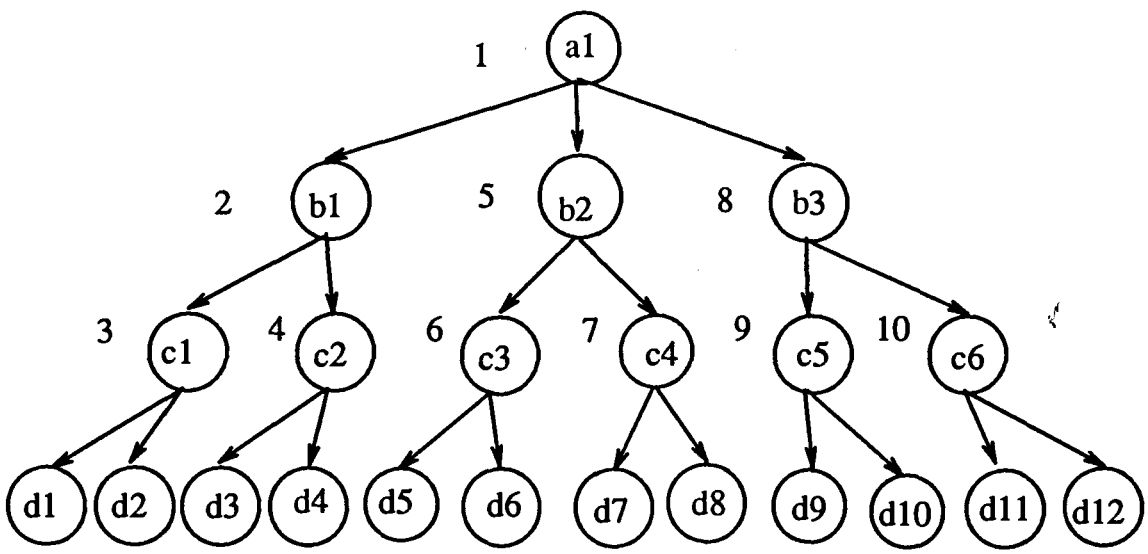


Figure 6-15: Closure Check Using Depth-First Search

the entire branch (meaning, all layers are generated if a mapping exists) emanated from that instance node, it examines *all* objects in the same layer before objects in the next lower layer are examined. However, root

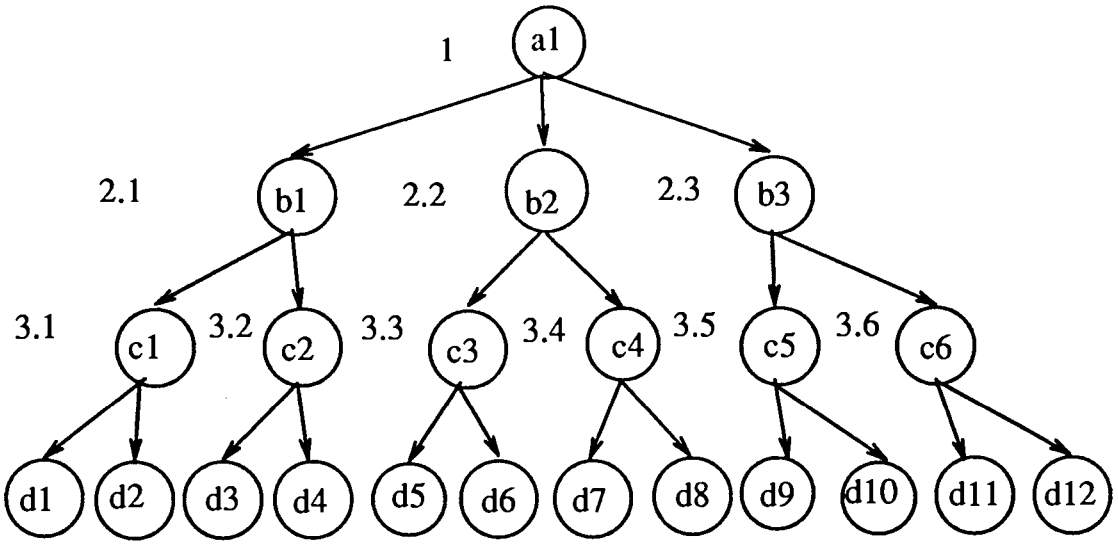


Figure 6-16: Closure Check Using Breadth-First Search

instances are always examined one instance at a time. Algorithm 2 shows the algorithm for the breadth-first search strategy.

Algorithm 2: Breadth-First Search Technique

a is given as the target node and the root node in the generation of the search trees. There are *j* functions in the chain. The *sizeof()* function returns the total number of elements in a set.

The variables used in this algorithm are as follows:

- *n1* is the total number of instances in object class *a*.
- *n2* is the number of output instances mapped by a single root instance a_{i1}.
- *n3* is the total number of output instances that are mapped to them by the input instance nodes which are immediately used as inputs to the function in the next **while**-loop.
- *n4* is the total number of instance nodes generated at the $(i+1)^{th}$ layer via the i^{th} function in the chain.
- *n5* is the total number of target instances which are returned as results to the chain.

- i is the index variable which controls iteration over all target instances returned as results to the chain.
- $i1$ is the index variable used to control iteration over all the instances of a.
- $i2$ is the index variable used to control iteration over $n2$ number of output instances being mapped to by the root instance or $n3$ number of output instances of an entire layer which are immediately used as inputs to the function, *Mapping*, to invoke the generation of the next layer of the search tree.
- $i3$ is the index variable used to control iteration over the output instance nodes generated by a single instance node.
- k is the index variable that determines the function to be applied at each layer during the generation of the search tree.
- f_k is the k^{th} function in the chain.
- *die* is a logical switch which is turned to TRUE if all the instance nodes at a certain layer of the search tree could not be mapped to even a single instance in the next layer of the search tree before the last layer of the search tree is generated.
- *pathlength* is the number of functions in the chain.
- *inputnode*, and *tempnode* are two-dimensional arrays used to hold the relevant target instances which are stored in the first column of the arrays, and the most recent instances generated for a certain layer of the search tree which are stored in the second column of the array.
- $outputnode_k$ represents the output node to be mapped to by the k^{th} function in the chain given an input node instance a_{i1} if the latter is a root instance, or *inputnode[i2, 2]* otherwise, at that invocation.
- *selectednode* is an one-dimensional array containing all the selected target instances which are results of the chain.

```
Program_Start()
begin
  n3 = 0
  for i1 = 1 to n1
     if there is a mapping f1(ai1) -> outputnode1
     then
        begin
           n2 = sizeof (outputnode1)
           for i2 = 1 to n2
              inputnode[i2, 1] = ai1
              inputnode[i2, 2] = outputnode1 [i2]
           endfor
        end

  if n3 > 0
  then
     begin
        die = FALSE
        k = 1
     end
  else
        die = TRUE

  while die = FALSE And k < pathlength
     k = k + 1
     n4 = 0
     n5 = 0
     for i2 = 1 to n3
        if there is a mapping fk(inputnode[i2, 2]) ->
             outputnodek
        then
           begin
              if k = pathlength
              then
                 begin
                    n5 = n5 + 1
                    selectednode[n5] = inputnode[i2, 1]
                 end
              else
                 for i3 = 1 to sizeof(outputnodek)
                   n4 = n4 + 1
                   tempnode[n4, 1] = inputnode[i2, 1]
                   tempnode[n4, 2] = outputnodek[i3]
                 endfor
           end
      endfor
  if n4 = 0
  then die = TRUE
  else
     begin
        inputnode = tempnode /*copy tempnode's content into inputnode*/
        n3 = n4
     end
 endwhile
```

```
if die = FALSE
then
    begin
      for i = 1 to n5
        output selectednode[i]
      endfor
    end
endfor
end
```

The major motivation of using depth-first search for closure check is in savings of space storage. Only the intermediate nodes generated in a branch under consideration are saved, whereas for breadth-first search, the space needed to store the intermediate nodes may grow exponentially as each layer of the search tree is generated. The number of objects that have to be kept track of by the system at any stage of processing using breadth-first search is always larger than that using depth-first search. However, the major drawback of closure check using depth-first search technique is in the interface cost. Because of the inherent nature of the functions in our data model, the parameter of a function could be a set, such that if depth-first search is used, one instance of the set at a particular layer *i* generates all layers of a branch in the search tree (if such a mapping exists) before the next instance in the set at that layer *i* is processed. However, this requires the constant interaction between the physical database and the function implementor in the architecture of our data model because every time a function is applied on an object, the set of instances being mapped to is returned to the function implementor for it to dispatch one of the instances from the set to the next function. Thus, the function routine invocation overhead will be high due to the switching of routines for each object being examined. Actually, the total number of function invocations for a search tree with *n* layers is equal to the total number of nodes generated in the first (n-1) layers. In contrast, minimal interface cost is the major motivation for the use of breadth-first search in closure check. Only one invocation of a particular function routine is needed to map *all* the objects at a certain layer of a search tree to *all* the relevant objects at the next lower layer, thus, there will be less interaction between the physical database and the function implementor. In other words, the function implementor gives a set of inputs instead of just one input object to the physical database in one function invocation and the physical database also returns a set of relevant output objects (usually object identifiers of these relevant output objects) for the entire layer of the search tree. Thus, there are only *n-1* function invocations for the generation of a breadth-first search tree with *n* layers.

Our chain processing method offers much more flexibility and efficiency than the use of nested-for loops [Dayal and Goodman 82] in processing queries. Let us use Example 6-6 to clarify this statement.

Example 6-6. Consider the QP graph shown in Figure 6-17 where *a* and *e* are the target nodes. This QP graph will be decomposed into two chains using our processing scheme, if *a* is chosen as the starting node of each chain. In that case, the chain f_1-f_2 can be evaluated using existence check. Here, a lot of computations can be saved because not all a_i instances in the object class *a* are involved in processing the other chain f_3-f_4. Only those relevant a_i instances which survive the f_1-f_2 chain will be considered.

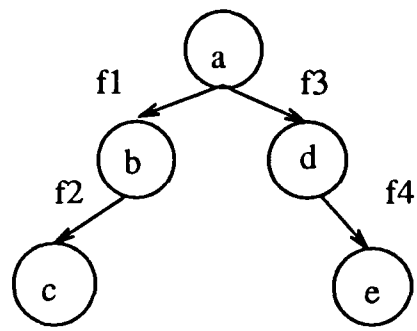


Figure 6-17: QP Graph for Example 6-6

Following the algorithm in [Dayal and Goodman 82], this QP graph is processed using a single *nested-for* loop as follows:

```
for each a
  for one b
  endfor

  for one c
  endfor

  for each d
    for each e
    endfor
  endfor
endfor
```

where *for one* refers to our existence check and *for each* refers to our closure check.

Using that technique, all a_i instances will be involved in the closure check for processing the chain f_3-f_4 because the outermost *for* loop is iterated over each a_i. This cannot benefit from the more restricted search space on *a* which could have been achieved if our chain processing method is used wherein only those a_i instances which satisfy the existence check of the chain f_1-f_2 will be considered.

6.2.5. Node Collapsing

Upon the identification of the processing unit which could be an edge or a chain (together with its searching techniques) in the QP graph, its constituent nodes are collapsed into a single node as their corresponding edges to which they are connected in that processing unit are processed.

Generally, nodes in a QP graph are collapsed in two ways, either by edges (functions) or by chains (groups of functions). The latter could further be classified as processing by simple chains which we call as **acyclic chain processing** and by cycles which is called **cyclic chain processing**. The remaining discussion will show clearly that there requires no special treatment in the processing of these two kinds of chains.

We will show how node collapsing is done, so that the resulting node can then be used for assembling with the remaining edges in the current QP graph at the next iteration of the query processing algorithm or returned as final results to the query if there are no more edges in the QP graph.

6.2.5.1. Edge Processing

Processing a function represented by an edge in a QP graph is seen as collapsing of nodes and/or boxes representing the non-primitive I/O object class parameters of that function into a single enclosing node. The edge just processed, together with the labeled box to which it is connected (if there are any) will be dissolved. All the nodes of the labeled box are also collapsed into the enclosing node. The enclosing node here represents a relationship object class which does not have a distinct user-assigned name, but is generated as a result of run-time computation as each function in the query is executed to keep track of the relevant objects that need to be considered for processing by the subsequent functions in the query. It is transient and is visible only to the system.

For example, after the execution of the function f(a) -> b where *a* and *b* are non-primitive object classes, the result will be tuple objects of $<a_i, b_i>$ where a_i and b_i components of each tuple object are of object classes *a* and *b* respectively that have a mapping to each other via the function *f*. This could be represented graphically as in Figure 6-18 wherein a bigger node encloses the component nodes *a* and *b* which represent the object classes of each of the components in the resulting tuple objects generated after the execution of that function. Upon the processing of the function *f*, the edge *f* is immediately dissolved from the QP graph.

Another example will be that of a labeled box connected to a node. Please refer to Figure 6-19. Note that prior to the execution of function f_1, the component nodes *a* and *b* in the labeled box are not related to each other since they are not of the same existing relationship object class but are of unrelated object classes.

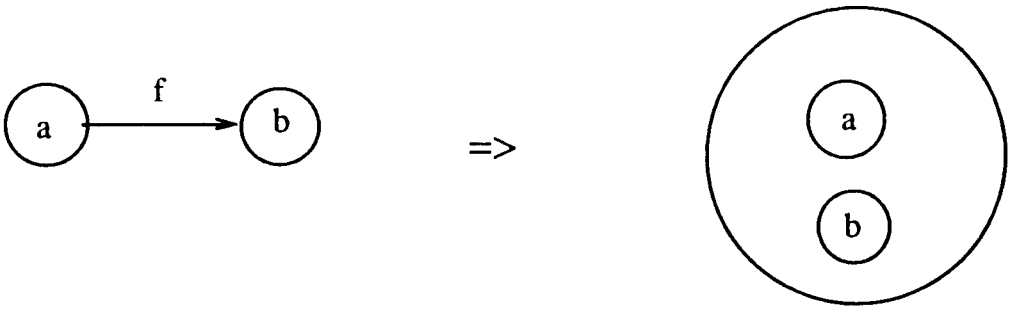


Figure 6-18: Processing by Edges

However, after the execution of the function, f_1(a, b) -> c where *a*, *b* and *c* are non-primitive classes, the nodes *a*, *b*, and *c* will then be collapsed into a single node and the labeled box and edge representing the function f_1 will be dissolved. At this point of time, instances from object classes *a*, *b* and *c* are related to one another via the function f_1 to form tuple objects $<a_i, b_i, c_i>$. As can be seen, the component nodes are now collapsed into a single enclosing node in the QP graph which instances are $<a_i, b_i, c_i>$ tuple objects.

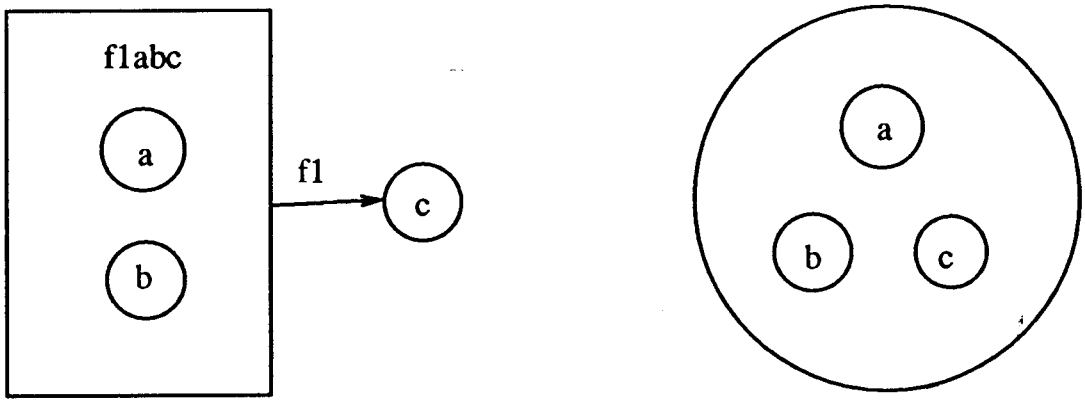


Figure 6-19: Processing by Edges

6.2.5.2. Chain Processing

Processing by chains is the processing of a group of two or more edges in the QP graph which are processed as a single unit that will eventually collapse the nodes connected by this group of edges into a single enclosing node after processing. Processing of chain is actually processing of cascaded functions wherein outputs of a function are used as immediate inputs to the next function in the chain such that only those relevant objects that could be mapped to by the previous function(s) are being considered by the next

function. This is true because all the functions in a chain are connected by the logical AND connective which simply implies a stronger restriction on the selected objects.

Example 6-7. Consider the functions in a simple chain (Figure 6-20):

f_1(a, e) -> b

f_2(b) -> c

f_3(c) -> d

where *a*, *b*, *c*, *d*, and *e* are non-primitive object classes. The QP graph representation will be a simple chain of three edges connecting the labeled box *f1aeb*, and nodes *b*, *c* and *d* together. Note that in this example, *a* and *e* are not component object classes of a common predefined or precomputed relationship object class, so they are enclosed by a labeled box instead of a node. The resulting enclosing node after the execution of this acyclic chain will consist of tuple objects <a_i, e_i, b_i, c_i, d_i> where a_i, e_i, b_i, c_i, and d_i are related to each other via functions f_1, f_2, and f_3. Thus, in QP graph representation, the component nodes of this enclosing node are *a*, *b*, *c*, *d*, and *e* to portray that the relevant tuple objects which satisfy the entire chain have been identified.

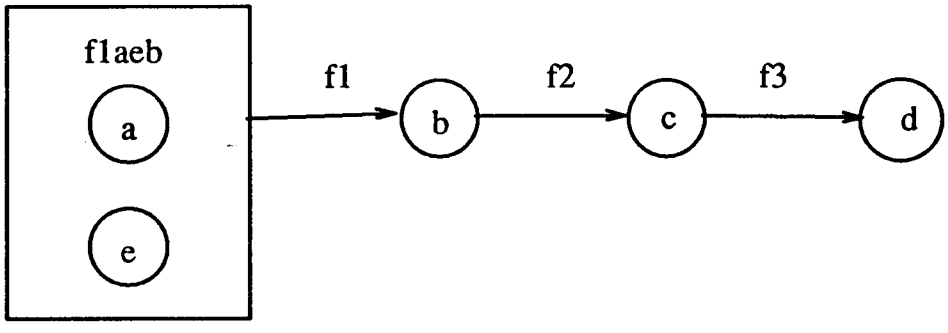


Figure 6-20: Acyclic Chain Processing for Example 6-7

Consider Figure 6-21 for the step by step graphical representation as this acyclic chain is processed. Upon the execution of function f_1, the edge f_1 and the labeled box *f1aeb* are dissolved. Note that the labeled box is dissolved because it is meaningful only prior to the execution of f_1 to show that *a* and *e* are the input object class parameters of the function f_1. After its execution, tuple objects <a_i, e_i, b_i> are formed where a_i, e_i, and b_i components in each tuple object are related to one another via the function f_1. This is represented graphically in Figure 6-21(1) where nodes *a*, *e* and *b* are collapsed into a single enclosing node. Upon the execution of the function f_2, new tuple objects <a_i, e_i, b_i, c_i> will be formed. Please refer to Figure 6-21(2).

Here, if b_i in <a_i, e_i, b_i> tuple objects of the enclosing node (a, e, b) shown in Figure 6-21(1) has a mapping to some c_i instances via function f_2, then <a_i, e_i, b_i> will be selected and collapsed with that c_i instance to form a single tuple object <a_i, e_i, b_i, c_i>. As could be seen, only b_i that could be found in the tuple objects of the enclosing node (a, e, b) generated by the previous function are being considered and checked for the existence of a mapping with the instances of object class c. In this way, the resulting effect of cascaded functions will be a restricted search space where only components of tuple objects that survive all the previously processed functions are considered for the processing of the subsequent functions instead of examining all the instances in a specified object class. The same explanation holds upon executing function f_3. Please refer to Figure 6-21(3). Here, if c_i in the tuple objects <a_i, e_i, b_i, c_i> of the enclosing node (a, e, b, c) found in Figure 6-21(2) has a mapping with d_i via function f_3, then the former tuple object <a_i, e_i, b_i, c_i> will be collapsed with d_i to form the new tuple object <a_i, e_i, b_i, c_i, d_i> in the new enclosing node (a, e, b, c, d). Notice that only the tuple objects that are generated as a result of processing some previous functions that *also* have a mapping with the currently processed function will be selected as components of the newly-generated tuple objects for the new enclosing node.

Now, let us examine the case of cyclic chain processing.

Example 6-8. Please refer to Figure 6-22. Given a cycle representing the functions: f_1(a) -> b, f_2(b) -> c and f_3(c) -> a, the step by step execution could be explained as follows: In step 1, upon executing function f_1, nodes a and b are collapsed into a single node as is shown in Figure 6-23(1). Here, the enclosing node consists of <a_i, b_i> tuple objects where a_i is mapped to b_i via the function f_1. All b_i's in the tuple objects <a_i, b_i> that can be mapped to some c_i instances via function f_2 will then be collapsed into a single tuple object <a_i, b_i, c_i> upon the execution of function f_2 in step 2 as is shown in Figure 6-23(2). In step 3, it is actually a selection of the tuple objects <a_i, b_i, c_i> generated in step 2 where c_i can also be mapped to a_i of the same tuple object via the function f_3. As can be seen, in cyclic chain processing, only relevant outputs of the previously processed functions are considered as inputs to the next function and the execution of the last function in the cycle should generate outputs which could also be found as components in its corresponding tuple object that is generated just prior to the execution of this last function.

In other words, cyclic chains are also cascaded functions which do not require any special way of processing that is different from acyclic chain processing.

We have presented processing by edges and by chains wherein after processing of these individual units, the result will be tuple objects which components are of object classes represented as nodes in that unit. In

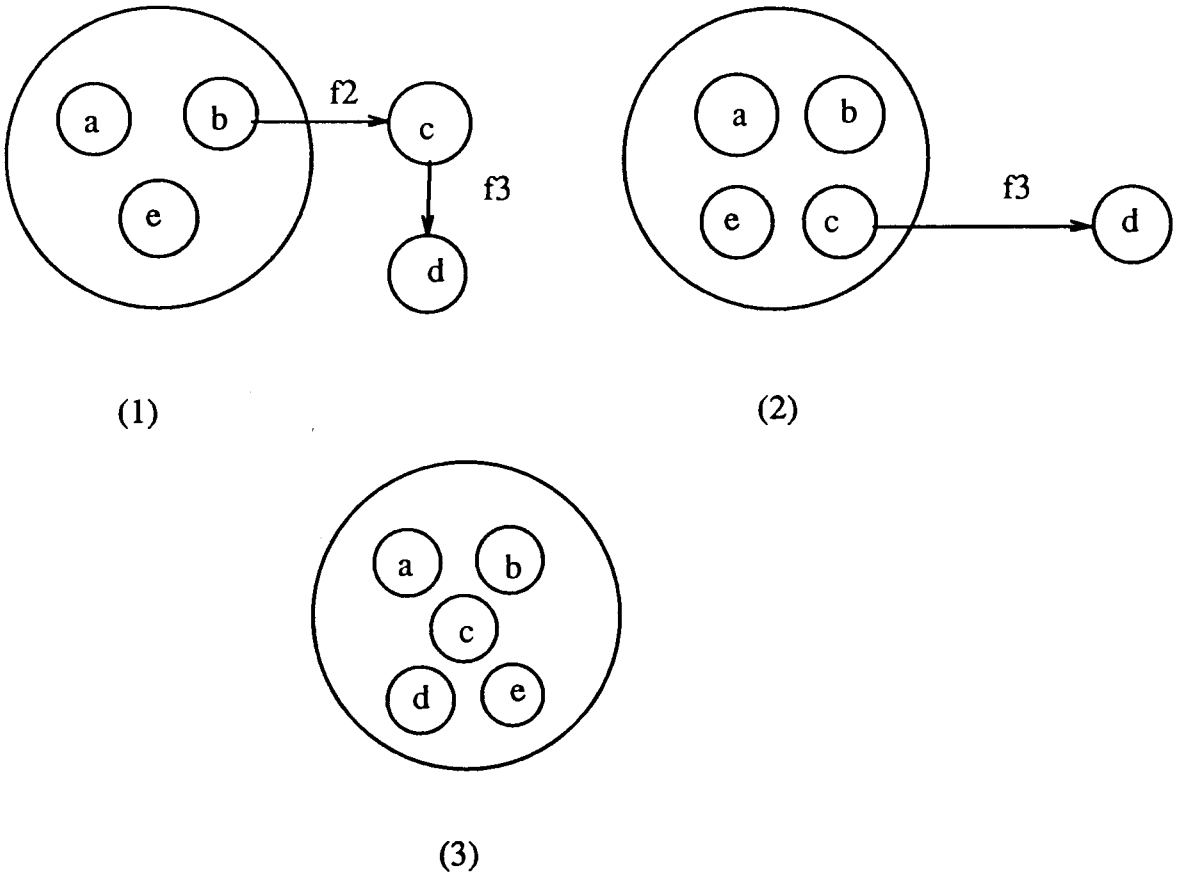


Figure 6-21: Processing QP Graph of Figure 6-20

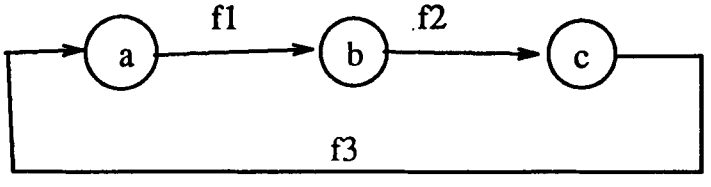


Figure 6-22: Cyclic Chain Processing for Example 6-8

reality, only the target nodes and the interface nodes are being kept track of by the system. Target nodes have to be kept track of because instances from these target object classes that satisfy all functions in the query have to be returned as results to the query. Interface nodes have to be kept track of also because the unprocessed edges to which they are connected may impose further restriction on the relevant tuple objects

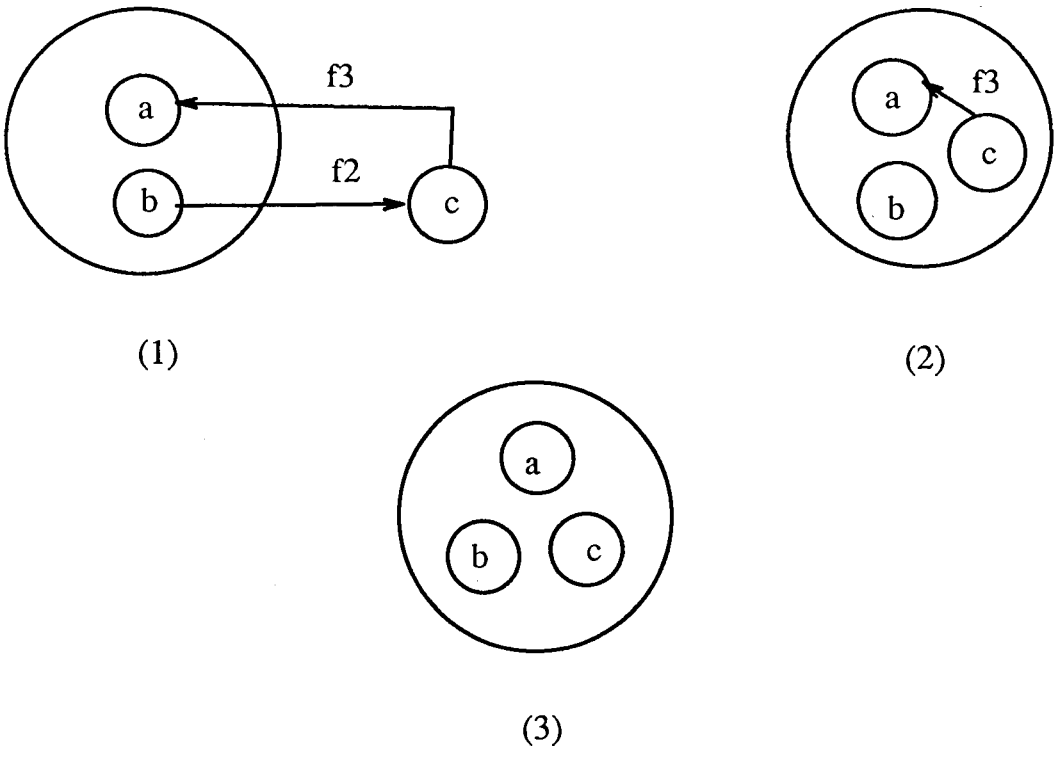


Figure 6-23: Processing of QP Graph of Figure 6-22

that can be considered for further processing by the subsequent functions. Thus, they may affect the target instances that can be eventually selected as results of a query. All other nodes that are neither target nodes nor interface nodes do not have to be kept track of because they will not affect the eventual target instances that can be returned as results since they are not used as parameters of any *unprocessed* functions.

Consider again Figure 6-11, where *e* is the target node and *a* is the starting node of the cycle f_1-f_2-f_3-f_4-f_5. Since nodes *b*, *c*, and *d* are all interface nodes which are part of some other chains other than the cycle f_1-f_2-f_3-f_4-f_5, then the processing of this cycle requires to keep track of all these nodes in the enclosing node. Another point is that since f_1-f_2-f_3-f_4-f_5 is a cycle, node *a* has to be kept track of also during the processing of edges f_1 through f_4. After processing the last edge f_5 of this cycle, each tuple object of the enclosing node will be of the form $<b_i, c_i, d_i, e_i>$. We can see that if we have a long chain where each node of the chain is an interface node that is connected to some other chains or edges, then the number of components in each tuple object after the processing of each edge in the chain will become bigger. This is in fact another disadvantage of having long chain with *many interface nodes* because the system needs to keep track of many interface

nodes aside from the absolute use of closure check for processing edges in a chain that are connected to interface nodes where the system cannot benefit from prunings of existence check.

Example 6-9. Let us round up the discussion of our query processing scheme with a query which involves both acyclic and cyclic chains. Assuming that all those non-relationship functions are processed prior to QP graph construction, the remaining relationship functions that need to be mapped into their corresponding QP graph representation are as follows:

f_1(b, c) -> d

f_2(d) -> b

f_3(a, b) -> e

f_4(e) -> f

f_5(f) -> g

f_6(g) -> h

f_7(h) -> a

f_8(j) -> k

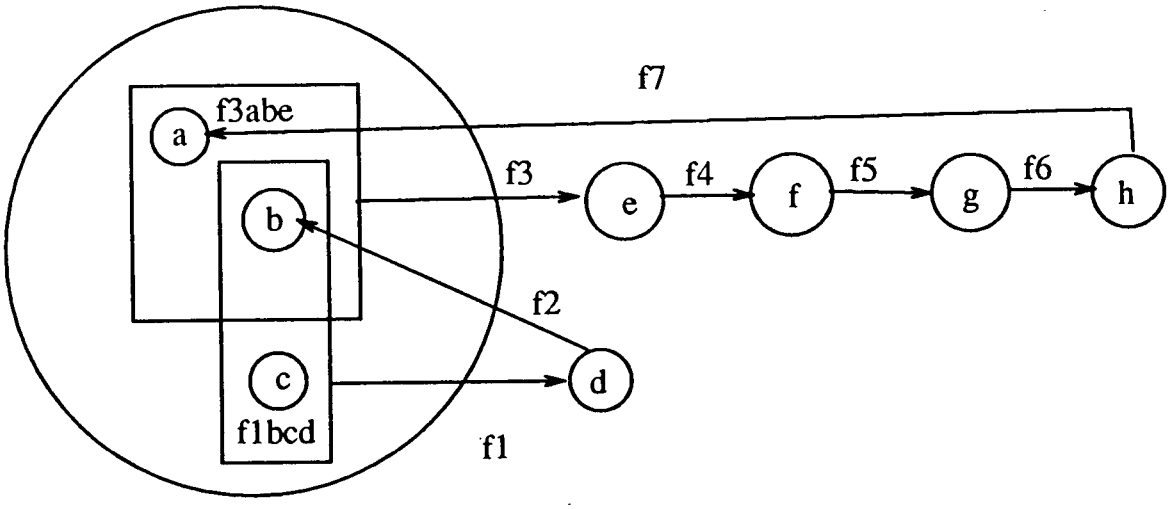
f_9(j) -> i

f_{10}(i) -> h

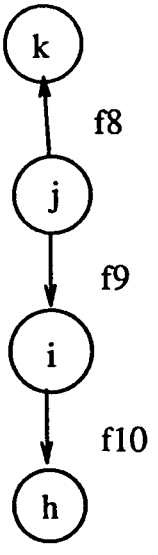
where *a*, *b* and *c* are non-primitive component object classes of a relationship object class generated as a result of the processing of a retrieval function f_0(a, b, c) -> s prior to the construction of this QP graph because *s* is its only output parameter and is of primitive object class. *e*, *f*, *g*, *h*, *i*, *j* and *k* are all distinct non-primitive object classes that are not related to each other prior to the construction of this QP graph.

Assuming node *g* is the target node, Figure 6-24 gives the QP graph after the query processor has identified the chains and edges and their processing sequence. The subscripts of the function labels in the figure indicate the sequence by which the edges are processed. We will show how QP graph is processed by showing the relevant nodes that are collapsed into a single enclosing node after the execution of each edge. Note that chains in QP graph of this example are not disjoint. They are drawn as separate chains for easy presentation purposes to show that each of them is processed as a single unit. In fact, Figure 6-24(1) has two cycles and Figure 6-24(2) is a simple chain. They all make up the complete QP graph of our query for this example. The following discussion will assume that only target node and interface nodes are being kept track of by the system as each edge is processed.

Figure 6-25 and Figure 6-26 show the collapsing of nodes as the two cycles in the first chain of the QP graph shown in Figure 6-24(1) are processed. The two cycles in this chain are f_1-f_2 and (f_1-f_2)-f_3-f_4-f_5-f_6-f_7



(1)



(2)

Figure 6-24: QP Graph for Example 6-9

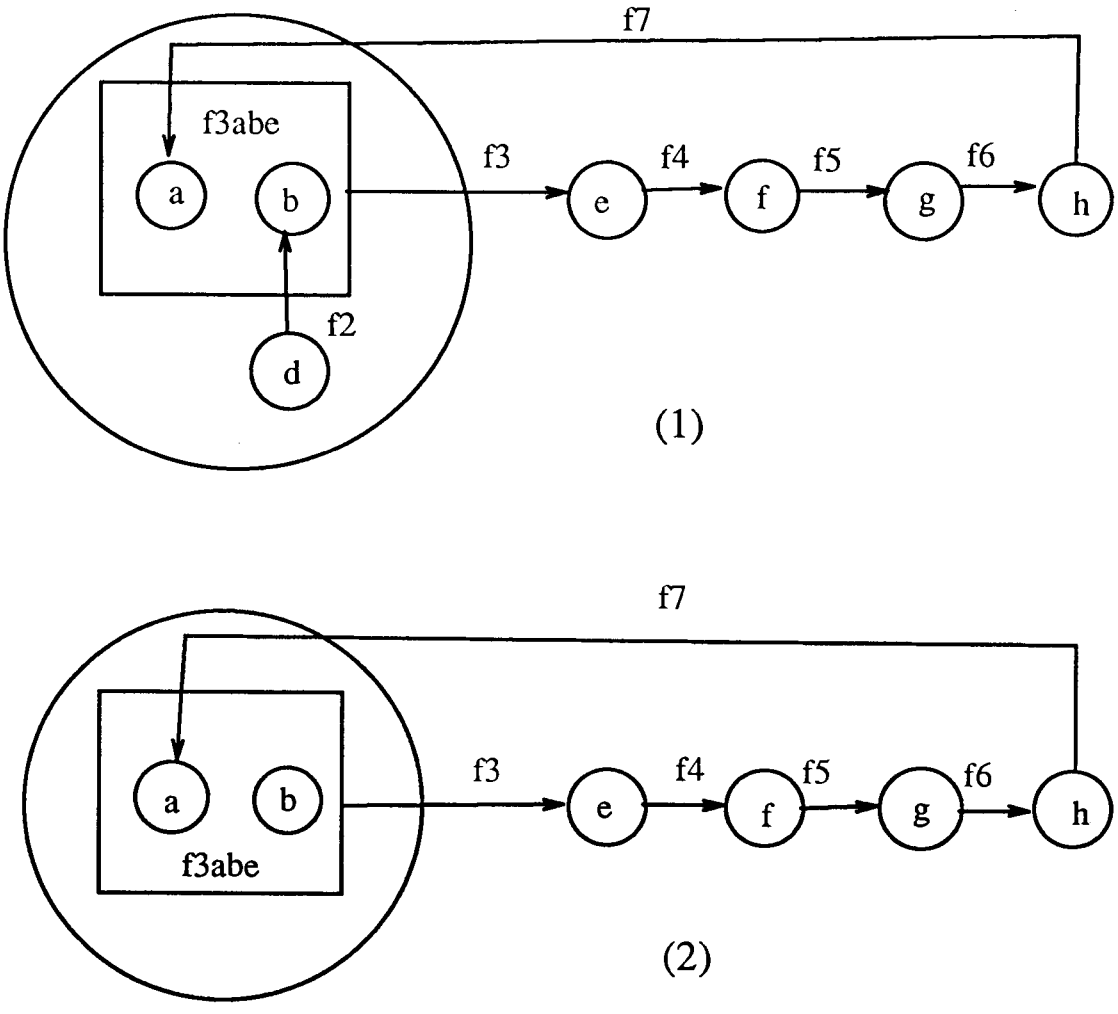
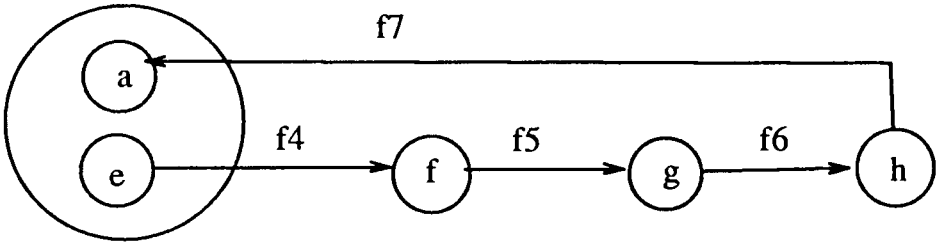
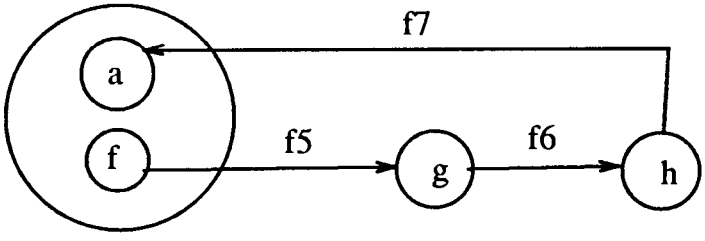


Figure 6-25: Processing of the Subcycle of the First Chain of Example 6-9

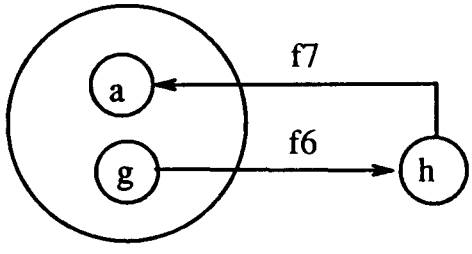
respectively where f_1-f_2 is a subcycle of the bigger cycle $(f_1$-$f_2)$-f_3-f_4-f_5-f_6-f_7. Note that the enclosing node with component nodes *a*, *b* and *c* tells us that the components a_i, b_i and c_i of each tuple object <a_i, b_i, c_i> are related to each other via the function f_0 which is executed prior to the construction of this QP graph. Step 1 collapses the former enclosing node with node *d* and produces tuple objects <a_i, b_i, d_i> where b_i, and c_i in the tuple object <a_i, b_i, c_i> of the former enclosing node in Figure 6-24(1) has a mapping with d_i via the function f_1. Note that *c* is not included as a component node in the new enclosing node shown in Figure 6-25(1) because *c* is no longer an interface node after the execution of the function f_1. Note that the edge representing function f_1 is dissolved after its execution and the labeled box *f1bcd* is also dissolved. Another



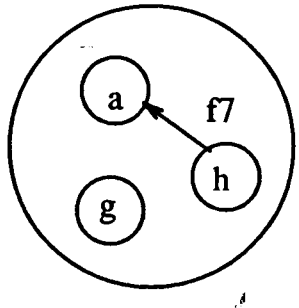
(3)



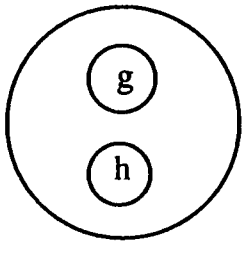
(4)



(5)



(6)



(7)

Figure 6-26: Processing of the Outer Cycle of the First Chain of Example 6-9

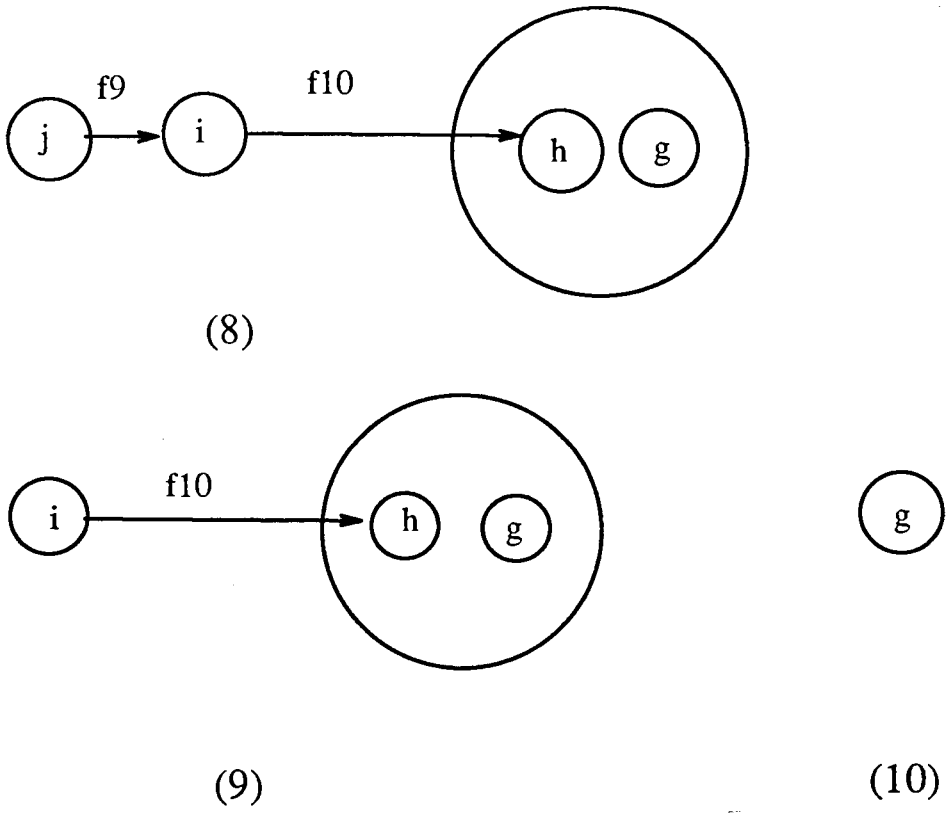


Figure 6-27: Processing of the Second Chain of Example 6-9

point is that when collapsing node *d* into the new enclosing node, *d* should not fall inside the labeled box *f3abe* because it is not one of the input parameters of the function f_3. In step 2, a_i, b_i in each tuple object $<a_i, b_i, d_i>$ of the enclosing node (a, b, d) of step 1 shown in Figure 6-25(1) will be selected if d_i can be mapped to b_i in that same tuple object. The resulting enclosing node will be (a, b) after the execution of the function f_2 with tuple objects $<a_i, b_i>$ as is shown in Figure 6-25(2). Now, we come to the processing of the outer cycle. Upon executing function f_3, $<a_i, e_i>$ tuple objects will be produced as is shown in Figure 6-26(3). Node *b* is not included as a component node in this new enclosing node because it is neither a target node nor an interface node after processing f_3. Execution of the functions f_4 through f_6 is simply processing of an acyclic chain which results of processing are shown in Figure 6-26(4) through Figure 6-26(6) respectively. Upon executing the function f_7, the component nodes in the enclosing node are just *g* and *h* as is shown in Figure 6-26(7). Node *g* is being kept track of by the system because it is the target node, while *h* is also being kept track of because it is an interface node that is connected to node *i* via the unprocessed edge f_{10}.

Both of them have to be kept track of for future processing of the remaining functions. Finally, the query processor proceeds with the processing of the simple chain of the QP graph shown in Figure 6-24(2). The node j in Figure 6-27(8) includes all those j_i objects which have a mapping with some k_i objects via the function f_g. The remaining processing steps are self-explanatory and are illustrated in Figure 6-27(9) and Figure 6-27(10). Note that eventually, node g will consist of all g_i instances that satisfy all functions of the query which are returned as the final results. We have shown in this section that upon the processing of each edge in a QP graph, it will be dissolved and nodes will be collapsed. This assures that each edge in a QP graph is processed once and only once and that the processing of the QP graph terminates when there are no more edges.

Chapter 7
Future Research Directions

There are still a lot of interesting issues in our proposed bi-level object-oriented data model and its query processing and optimization strategies which require further studies to make our design into a full-fledged system. In particular, there is a number of things that need immediate attention.

First of all, we need a better idea of how the different abstraction hierarchies could be applied in the geometric object data model. For example, the NODE object class could be related to the POINT object class via the IS-A class hierarchy. We may even have line segments to represent a portion of a line, in which case, the two are related to each other via the PART-OF abstraction hierarchy. The abstraction hierarchies among geometric objects have to be clearly defined to facilitate the actual implementation of the geometric functions.

Second, we have shown the usefulness of superfunctions and the OFQL user-interface in the geographic object data model. In fact, these two interfaces will also be very useful in the geometric object data model. If we consider the data entry and editing component of a GIS system, there are a lot of superfunctions which are purely defined in terms of geometric functions and geometric objects. Some of the examples are SNAP, EDGEMATCH, DISSOLVE, and other spatial functions that are needed in the input and update process of spatial data in the physical database. We call these functions as the *input functions*. These input functions could be thought of as superfunctions at the geometric object data model which implementation may involve some existing geometric functions together with some other computations. OFQL interface at the geometric object data model is only needed and used by the people responsible for the building and maintenance of spatial data in the physical database. Our thesis limits the scope of study and discussion on the application of the different modeling constructs, abstractions and interfaces to the geographic object data model only, but a gentle touch on the surface does tell us that majority of them could also be applicable to the geometric object data model.

The difference between superfunctions defined at the geometric object data model and those geometric functions (which are also defined in terms of other geometric functions and/or together with some additional

computations) that are being mapped to by the semantic spatial functions at the geographic object data model is obvious. *Geometric functions* refer to the set of spatial functions at the geometric object data model that is used to implement or define the set of semantic spatial functions at the geographic object data model. Thus, it must be adequate to handle all the semantic spatial functions. *Superfunctions* defined at the geometric object data model, on the other hand, are never mapped to by the semantic spatial functions in the geographic object data model. They are used to manipulate the input and update process of the geometric objects.

Thirdly, a possible extension on OFQL to incorporate universal and existential quantifiers on the set values of the parameters of the functions will allow user to pose queries with a more precise condition specification imposed on the geographic objects. In the discussion of this thesis, a target object is selected as long as there *exists* an instance that could be mapped to all the predicates of the query and make all of them true. The future study will be to allow user to write queries that can express which predicate requires *all* or a specific number of instances of a particular object class that is used as its parameter to be evaluated to true in order for it to be considered as a valid mapping. The aggregate functions like Count, Sum and others may require further study for their incorporation into the OFQL user interface. Their query processing issues should be examined also if a user is interested to apply such functions on objects which are not specified after the SELECT clause. Query processing and optimization strategies need further study to handle negation, quantifiers and predicates that are connected by the logical OR connective.

Finally, temporal aspect demands research attention to handle changes in both spatial and non-spatial data of all geographic objects over any time frames. This will allow us to incorporate the management of temporal data and its association with the spatial and non-spatial data in the database. An appropriate way of incorporating temporal data into the bi-level object-oriented data model should be found.

Chapter 8
Conclusion

In this thesis, we have introduced a bi-level object-oriented data model with a function form of interface called OFQL which is appropriate for applications like GIS. We have described the basic modeling constructs for each layer of our bi-level data model which include the geographic and the geometric object data models. An overall architecture of a GIS system using our bi-level data modeling approach is presented. Some query processing and optimization issues of OFQL are also discussed. In particular, we consider the following as the major contributions of this thesis:

First, we have introduced a bi-level object-oriented data model that is appropriate for applications like GIS. It provides abstraction and functional independence between the geographic object data model and the geometric object data model where each has its own set of object classes, objects and functions.

Second, we have designed high-level user interfaces which include the function interface using pointing devices and the high-level query language OFQL. Function models well run-time computations, attributes of objects and relationships among objects. It serves as an explicit mechanism that models the topological or spatial relationships among geographic objects. Furthermore, it provides a set of basic tools from which user can build more application-domain-specific functions and queries, thus, enhancing system extensibility. With this high-level interface, coupled with our bi-level approach, the actual spatial representation of geographic objects and the implementation details of the behavioral aspects of geographic objects are transparent to the user while he can pose both spatial and non-spatial queries of a geographic nature.

Third, with the proposed architecture of our GIS system, the query processor does the automatic mapping between the two layers of the bi-level data model and the function implementor does the necessary interfaces with the underlying database.

Finally, different query processing and optimization issues of OFQL have been discussed. In our query processing scheme, IS-A and PART-OF abstraction hierarchies are resolved prior to QP graph construction which simplifies subsequent query processing. We have processing by chains aside from the conventional

way of processing by edges. This provides early restriction on search space and allows possible application of existence check and depth-first search technique in processing chains. Depth-first search is particularly attractive in an object-oriented environment where object clustering and object caching are supported. Furthermore, our query processing scheme can handle more than one target object class in a QP graph. Cyclic query (a query which functions are processed as a single cyclic chain) processing, acyclic query (a query which functions are processed as a single acyclic chain) processing, and processing of a single query consisting of a mixture of both cyclic and acyclic chains are handled in the same way as a direct consequence of the concept of cascaded function execution.

References

[Banerjee et al. 87]
Banerjee, J. et al.
Data Model Issues for Object-Oriented Applications.
ACM Transactions on Office Information Systems 5(1):445-456, 1987.

[Bretl et al. 89] Bretl, R. et al.
The GemStone Data Management System.
Object-Oriented Concepts, Databases, and Applications.
New York: ACM Press, 1989.

[Dayal and Goodman 82]
Dayal, U. and Goodman, N.
Query Optimization For CODASYL Database Systems.
In *ACM SIGMOD International Conference on Management of Data '82 Proceedings*, pages 138-150. Orlando, Florida, 1982.

[ESRI] *ARC/INFO Users Manual Version 3.0, The Geographic Information Systems Software*
ESRI, .

[Feuchtwanger 87]
Feuchtwanger, Martin.
An Object-Oriented Semantic Association Model for GIS.
In *GIS National Conference.* Ottawa, 1987.

[Fish et al. 87] Fishman, D. H. et al.
Iris: An Object-Oriented Database Management System.
ACM Transactions on Office Information Systems 5(1):48-69, 1987.

[Guttman, A. 84] Guttman, A.
R-trees: A dynamic index structure for spatial searching.
In *Proceedings of the SIGMOD Conference*, pages 47-57. Boston, 1984.

[Heiler and Zdonik 88a]
Heiler, S. and Zdonik, S.
FUGUE: a Model for Engineering Information Systems and Other Baroque Applications.
In *Proceedings of the Third International Conference on Data and Knowledge Bases.* Jerusalem, Israel, 1988.

[Heiler and Zdonik 88b]
Heiler, S. and Zdonik, S.
Views, Data Abstraction, and Inheritance in the FUGUE Data Model.
Advances in Object-Oriented Database Systems.
Springer-Verlag, 1988, pages 225-241.

[Kim 90] Kim, Kyung-Chang.
Parallelism in Object-Oriented Query Processing.
In *IEEE Proceedings 6th International Conference on Data Engineering*, pages 209-217. 1990.

[Kim, Bertino and Garza 89]
Kim, W., Bertino, E. and Garza, J.
Composite Objects Revisited.
In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 337-347. 1989.

[Kim, Kim and Dale 88]
Kim, K.C., Kim, W. and Dale, A.
Acyclic Query Processing in Object-Oriented Databases.
In *Proceedings 7th International Conference on E-R Approach.* 1988.

[Kim, Kim and Dale 89]
Kim, K.C., Kim, W. and Dale, A.
Cyclic Query Processing in Object-Oriented Databases.
In *IEEE Proceedings 5th International Conference on Data Engineering*, pages 564-571. 1989.

[Ooi, Sacks-Davis and McDonell 89]
Ooi, B. C., Sacks-Davis, R. and McDonell, K.
Extending a DBMS for Geographic Applications.
In *IEEE Proceedings 5th International Conference on Data Engineering*, pages 590-596. Portland, Oregon, 1989.

[Rowe 87] Rowe, Lawrence.
The POSTGRES Data Model.
In *Proceedings of the 13th VLDB Conference*, pages 83-96. Brighton, 1987.

[Samet, H. 84] Samet, H.
The Quadtree and Related Hierarchical Data Structures.
ACM Computing Surveys 16(2):187-260, 1984.

[Samet, H. 88] Samet, H.
Hierarchical Representations of Collections of Small Rectangles.
ACM Computing Surveys 20(4):271-309, 1988.

[Smith and Zdonik 87]
Smith, Karen and Zdonik Stanley.
Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems.
In *OOPSLA '87 Proceedings*, pages 452-465. 1987.

[Star and Estes 90]
Star, Jeffrey and John Estes.
Geographic Information Systems An Introduction.
Prentice Hall Inc., 1990.

[Su, Krishnamurthy and Lam 89]
Su, S.Y.W., Krishnamurthy, V. and Lam.
An Object-oriented Semantic Association Mocel (OSAM*).
AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications.
American Institute of Industrial Engineering, 1989.

[Weiser and Lochovsky 89]
Weiser, Stephen and Frederick Lochovsky.
OZ+: An Object-Oriented Database System.
Object-Oriented Concepts, Databases, and Applications.
New York: ACM Press, 1989.

[Wilkinson, Lyngbaek and Hasan 90]
Wilkinson, K., Lyngbaek, P., and Hasan, W.
The Iris Architecture and Implementation.
IEEE Transactions on Knowledge and Data Engineering 2(1):63-75, 1990.