QUERY OPTIMIZATIONS FOR A SPATIOTEMPORAL QUERY PROCESSING SYSTEM

by

Charlie Chengyu Huang

B. Eng., Tsinghua University, China, 1990

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

APPROVAL

Name:	Charlie Chengyu Huang
Degree:	Master of Science
Title of thesis:	Query Optimizations for a Spatiotemporal Query Processing System

Examining Committee: Dr. Stella Atkins
Chair

Dr. Wo-Shun Luk
Professor, Computing Science
Senior Supervisor

Dr. Jiawei Han
Associate Professor, Computing Science
Supervisor

Dr. Raymond Ng
Assistant Professor, Dept. of Computer Science,
UBC
External Examiner

Date Approved: March 25, 1994

To Baba and Mama

Abstract

Database front-end systems and Application Specific Query Languages are commonly used for various applications to best fit in users's special needs and performance requirements. A front-end system, the VPD system, is presented in this thesis. It is created for the Vancouver Police Department to query and display the crime distribution in the city of Vancouver. Determined by its usage, the VPD system mainly features spatiotemporal queries. The query language proposed for the VPD system is called ESQL (Extended SQL). This thesis concentrates on query optimizations for the VPD system.

The VPD system is implemented on both Relational DBMS Sybase and Object-oriented DBMS Objectstore. Query performances on these two platforms are compared and factors affecting the performances are found out through a series of systematic experiments and penetrating analysis. Certain techniques are applied to improve the VPD system's performance for processing general queries on Objectstore DBMS. In order to further enhance the speed of processing spatiotemporal queries, a multi-key index tree, the $KDB^+ - tree$ is proposed. Based on the $KDB^+ - tree$ indexing mechanism, the Spatiotemporal Query Processor (STQP) is introduced to the VPD system. The STQP can process any ESQL query yet favors spatiotemporal queries. It makes use of all the query optimization techniques proposed in the thesis and provides efficient query processing for the VPD system.

Acknowledgements

First and foremost, I would like to thank my senior supervisor, Dr. Wo-Shun Luk, for his patient supervision and constant support. His encouragement and advice helped me to go through all the tough times of my research.

I am grateful to Dr. Jiawei Han for his great help throughout my study and serving on my supervisory committee. I also want to thank Dr. Raymond Ng for graciously consenting to be my external examiner and Dr. Stella Atkins for serving as the chairman of the examining committee.

I would like to thank all my friends who have provided me with their generous help. Special thanks go to Craig Jones and Ju Wu for their help in the beginning of my project, Wei Zhou for her cooperation to my work, Gabor Melli for his reliable system support, Graham Finlayson, Daya Gaur, Chor Guan Teo and Narayan Vikas who spent their precious time reviewing my thesis draft and giving me valuable comments.

Finally, I would like to thank my parents for their greatest love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The VPD System

The research in this thesis grows out from the development of a practical project. For this project, we created a specialized GIS (Geographic Information System) front-end system to query and display the crime distribution in the city of Vancouver. As this system is implemented for the Vancouver Police Department, we name it the VPD system. The data used for the VPD system enclose the information on crime incidents in Vancouver, including the type, location, time of the crime, etc. This set of data are all point data. The spatial attributes and temporal attributes in this data set represent points (geographical points and time points), instead of complex data structures, such as polygons or time intervals.

Most of the desired queries would involve query constraints on spatial and temporal information. An example of a spatial query is to investigate all the crimes that occurred in a certain area, such as a rectangle, a circle, a buffer along a street or even an irregular-shaped area. An example of a temporal query is to inquire information on the crimes that occurred between every Friday 6:00pm to Saturday 8:00am from Jan. 15, 1991 to Feb. 22, 1991. In most cases, the user would query over both spatial and temporal information. We call such a query a *spatiotemporal* query. Of course, the query may also involve other non-spatiotemporal information, like the crime type, etc. In the VPD system, spatiotemporal queries account for the major part of our users' need and, as I just mentioned, all these queries are to be performed over spatiotemporal point data, which are plain record data.

In order to express these special purpose queries in a very concise and user-friendly way so that it would be easy and simple for users to use, especially for non-computer professionals, and also to make the system most efficient towards processing spatiotemporal queries, we believe that to build an application-specific query processing system is the best way to achieve the goal. This system would be a database system front-end on top of an existing Database Management System (DBMS). A front-end usually refers to an interface layer on top of a DBMS engine to provide the user with more convenient and efficient query facilities. Almost every front-end has its own query language. A front-end and its underlying DBMS form a virtually new DBMS from the user's point of view. In the VPD front-end system, a special type of query language, ESQL (Extended Structured Query Language), is designed as a CQL (Customized Query Language) to facilitate users with powerful query expression. The language's name comes from that it has extended ability to deal with spatial and temporal queries while keeping the basic features of SQL at the same time.

The query performance is always very important for a front-end system. The VPD system is implemented on both object-oriented DBMS (OODBMS) and relational DBMS (RDBMS) platforms, which are Objectstore and Sybase in this case, respectively. Through the performance comparison and analysis of the speed-affecting factors, a series of query optimization techniques are developed to improve our system's performance, especially towards the spatiotemporal queries.

1.2 Related Work

My research is closely related to work done in two major database sub-areas: query language front-end and query optimization of spatiotemporal queries on OODBMS and RDBMS. A database query language is a data manipulation language for end users to retrieve relevant information from databases. In most cases, it is a non-procedural language. Its basic tasks include selection, projection, join, etc. Originally developed in late 1970's by IBM's San Jose Research Laboratory, SQL (Structured

Query Language) has been successfully adopted by many database management systems and been approved as a standard query language by ANSI (American National Standards Institute).

However, with the development of database systems, especially with the fast evolution of object-oriented concepts into DBMS, there are many cases when it is impossible or inconvenient to use SQL to do the query. Recently, a lot of studies and research have been done on designing and implementing various kinds of query languages and their corresponding front-ends to fit in different application requirements. Some of them are constructed as one part of general DBMSs. For instance, the object-oriented database system ORION developed by Dr. Won Kim's group[Bane88][Kim89] has its query language suitable for object-oriented queries. Some other systems build their own front-ends based on existing DBMSs. For instance, W. Meng et al[Meng93] introduced a relational front-end for an object-oriented database system. Interestingly, at the same time, V. Markowitz and A. Shoshani presented their object-oriented query front-end over relational DBMS and SQL. These systems are designed for certain circumstances where the application can utilize both the expressive power of the front-ends and the processing power of the underlying DBMSs.

Besides the above relatively general-purpose systems, there also exist a large number of systems which are designed mainly for processing specific applications. In these systems, a CQL or so-called ASQL (Application-Specific Query Language) is needed. Just in the area of processing spatial and temporal queries, a lot of research and work have been done to develop various ASQLs. In [Aref91], W. Aref and H. Samet developed their SAND (Spatial And Non-spatial Data) spatial data model and corresponding query language for spatial query. R. Snodgrass[Snod87] proposed TQuel to handle temporal data and queries by extending the standard query language Quel with some temporal features in Ingres. J. Wu[Wu92] introduced the "Block World" model and the DSQL (Dynamic Spatial Query Language) to execute 3-D spatial queries. W. Kafer and H. Schoning presented their MQL language for temporal queries upon their TMAD database model. X. Xu[Xu90] presented her STSQL query language for spatiotemporal queries. Furthermore, almost all of these spatial/temporal systems consist of certain query optimization methods suitable for their special applications.

All these query language front-ends introduced before have very similar structures. They are all made up of their own query languages, query parsers to parse these queries and query processors to execute queries. For most of them, various query optimization techniques are utilized to improve processing performance of their specific queries. Most importantly, the purposes of these front-ends are the same: to have a combination of the expressive query language and powerful query processor. All the previous work has inspired us to create our application-specific front-end system, the VPD system, and to develop certain optimization techniques to improve its spatiotemporal query performance.

At present, object-oriented DBMS and relational DBMS are the two dominating database system branches. Most Database Management Systems (DBMSs) fall into these two categories. While more and more research and studies have applied object-oriented technology to database systems, object-oriented DBMSs are enjoying a great success in areas like CAD and CAM when processing large and complicated data, such as geometric elements like polygons, audio and video data, text and pixel map, etc. Quite a few of previous studies have demonstrated the efficiency of OODBMS by doing performance comparisons between OODBMS and RDBMS. R. Cattel[Catt91] used the benchmark OO1 (Object Operations version 1) to show the performance superiority of an OODBMS against a RDBMS but he did not not disclose the names of these DBMSs. J. Wu[Wu92] and W. Zhou[Zhou93] also made use of Objectstore and Sybase, the same DBMS platforms as used by VPD system, to do the performance tests. They both showed that Objectstore has a better performance when handling complex data objects through navigational data access (to access data by pointers).

Nevertheless, RDBMSs are still used by the majority of DBMS users whose data being processed are "ordinary" plain record data, as what happens in most business and office data processing. This is because that most OODBMSs still do not seem to be so efficient as RDBMSs when doing associative queries on set-oriented data. What puzzles us is that since object-oriented technology still supports simple formatted record data well, why are their performances not very satisfying? What are the factors determining an OODBMS' performance, the way it organizes data, its architecture or something else? Driven by these questions, we test and compare the performances of VPD system on OODBMS and RDBMS, using associative queries on plain record data. A series of experiments and research lead us to the secrets behind the performance and some effective solutions to improve the query performance.

1.3 Objectives

Query performance is a fairly practical issue and is determined by many factors. The goal of this thesis is to find these factors from experimental analysis and apply appropriate adjustments to them so as to optimize query performance. Because Sybase does not provide users with control over single data items and it makes use of a server-process-all client/server architecture, unfortunately, most of the query optimization techniques stated in this thesis can be only applied on Objectstore platform. We just use the VPD system's Sybase performance as a reference to its query performance on Objectstore.

The optimization methods for the VDP system are developed by two steps. The first step is to improve Objectstore's general query performance. Both Objectstore's internal query strategy and external architecture are studied. Internally, Objectstore's query facility is still at an immature stage. It does not use the right strategy to utilize indexes. Neither is its retrieving method satisfactory. Index-rearranging and collection-sorting are adopted to overcome these Objectstore's defects to raise query speed. Externally, unlike Sybase, Objectstore relies on its client process and client cache to do all the processing, leaving its server only as a page server. In this case, choosing an appropriate client site configuration and managing to turn the client cache into a replicate database would greatly enhance the query performance.

The second step is taken toward the optimization of spatiotemporal queries. Indexes and the way of using indexes are very crucial to performance. A multi-key index on the commonly used spatial and temporal attributes would be the right approach to use. A lot of multi-dimensional index structures have been proposed, such as Bentley's KD-tree[Bent75], Samet's quatree[Same84], Robinson's KDB-tree[Robi81]. However, as neither of them can well fit in the VPD system, a new index structure, the $KDB^+ - tree$, derived from the KDB-tree, is presented. The use of the $KDB^+ - tree$

further improves the VPD system performance towards sptatiotemporal queries and also partly eliminates the impact from Objectstore's index problem.

Through this thesis, we show that from the performance's point of view, the architecture layout of a DBMS could be as important as the internal query processing method. Both of them are crucial for a good query speed. On the other hand, the way how to organize data, either through a table as in RDBMS or through a collection of pointers as in OODBMS, may not help to determine the performance. This may be well against the general intuition. However, since the object-oriented data model give users more control over each data object, it leaves us more room for utilizing various optimization techniques. Making appropriate use of special properties of the underlying DBMS can lead to potential query processing optimizations of a database front-end. This would be a very useful hint for future query optimization if the outlined techniques cannot be reused directly.

1.4 Outline

The organization of the rest of the thesis is stated in the following. First, the structure and all the components of our VPD system will be discussed in Chapter 2, including the Graphical User Interface (GUI), the Query Parser, the Query Processor and the databases being used. Also we would introduce the sample queries and system configurations being used for our experiments. In chapter 3, we will be focusing on performance comparisons of the VPD system on Sybase and Objectstore. By analyzing a series of experiments on our sample queries, the major factors that affect the Objectstore associative query performance on flat record data will be discovered. By adjusting these factors, a much better performance will be achieved at the end of the chapter.

However, the index problem is only temporarily solved in chapter 3. A multi-key index tree, the $KDB^+ - tree$ proposed by us will be presented in chapter 4 with its advantage in multidimensional queries over the KDB-tree, from which the $KDB^+ - tree$ is derived. The superior performance of the $KDB^+ - tree$ for spatiotemporal query processing will be demonstrated in chapter 5. Also in chapter 5, a sketch design of the Spatiotemporal Query Processor, which is in favor of processing spatiotemporal queries, will be presented. Simulated experiments will be given to illustrate the high speed of this processor. Finally, the conclusion and future work will be discussed in Chapter 6.

Chapter 2

The VPD System

The design and implementation of our VPD system is described in this chapter. This system is created for the Vancouver Police Department to query the crime rates in a specified region and within a certain time period. The VPD system consists of 4 major parts, the GUI, Query Parser, Query Processor and the Databases.

2.1 Overview of the VPD System

The VPD system is composed of 4 major parts: Graphical User Interface (GUI), Query Parser, Query Processor and Databases, as shown in Fig. 2.1. The GUI is responsible for taking the queries input from the user and displaying their results. When the VPD system is started, the GUI will consult the Databases to get the geographical data for map of Vancouver and the boundaries for *enumeration areas.* An *enumeration area* (*EA*) is the smallest census survey unit according to Statistic Canada. Then the GUI displays the map and these about 800 *EAs.* On top of the map, a series of menu buttons are displayed for user to operate the VPD system. Some snapshots of the GUI can be found in Appendix A. The user can either type in his ESQL query in a pop-up window or use a combination of menus to input query constraints separately and let the GUI form the ESQL query. The ESQL query will then be sent to the Query Parser. After the query results are sent back to the GUI, they are displayed on the Vancouver city map by coloring the map. Different colors are used to represent different crime rates in queried enumeration areas (This is why the projection predicate in our later sample queries is "EA"). This GUI is written in X window (Xt/Xlib), Open Look and HOOPS.

The Query Parser follows the BNF of the ESQL to parse ESQL queries input from the GUI. Once an ESQL query is parsed, its query information is retained and stored in the *parse tree*, which is a tree structure broadly used for storing query conditions. Different from some other parse trees[Oren92][Wu92], our VPD parse tree also contains extra structure to contain spatiotemporal query constraints besides the normal conjunction-disjunction tree. After the parsing finishes, the *parse tree* is passed to the Query Processor and provides it with all the necessary information and values for query processing, such as the projection list, the table (collection) name, the selection criteria and spatiotemporal conditions, etc. More details will be discussed in Section 2.3.

After the Query Processor receives the *parse tree* from the Query Parser, it will form actual DBMS queries utilizing the information stored in the *parse tree* and send these "real" queries to Databases. After the underlying DBMS executes these queries and retrieves information from databases, query results are sent back to Query Processor and then back to GUI for display. The design and implementation of the Query Processor relies on the underneath DBMS. Queries generated by Query Processor are subject to the DBMS being used. Even upon the same DBMS, different query approaches may also result in different implementations of the Query Processor. In this thesis, we present 3 versions of the Query Processor based on the DBMS platforms and query approaches being used:

- Utilizing SQL on Sybase platform.
- Utilizing Objectstore collection query facility on Objectstore platform.
- Utilizing self-designed query facility on Objectstore platform.

The design and implementation of the Query Processor will be further described in Section 2.4 and Chapter 5.

There are two major databases in the VPD system. One contains all the geographical data for the map of city of Vancouver and all the EA boundaries. These data are sent to GUI directly for drawing the user interface. The Query Processor does not need to access this database. The other database contains all the crime data, which will be used for executing queries from Query Processor. We mainly concern this database and when we talk about database in the rest of the thesis, we refer to this crime database. Two copies of the databases are created, one copy on top of Sybase and the other one on Objectstore platform. Both copies have identical contents, which will be talked about in next section. The structure of the whole system is shown in Fig. 2.1.

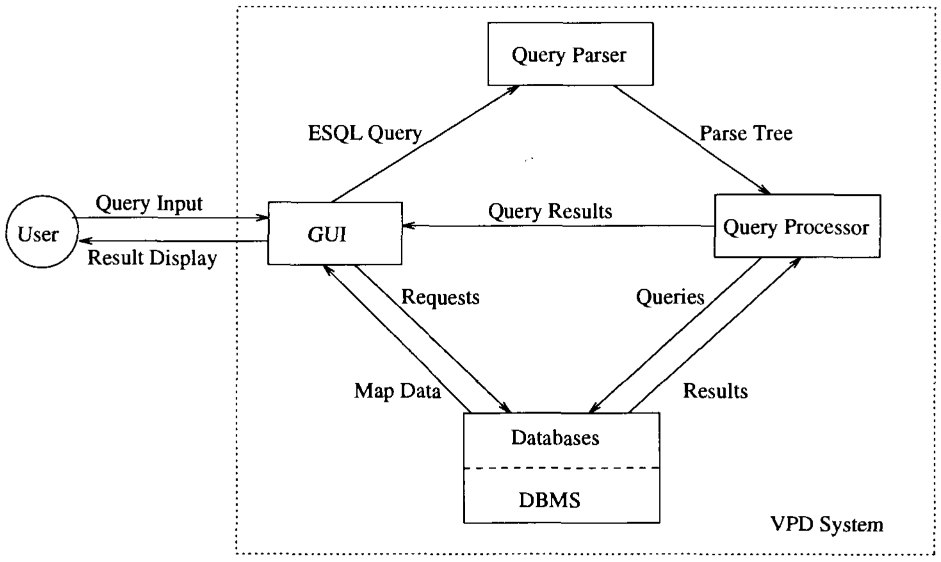


Figure 2.1: The Components of the VPD System

2.2 The VPD Crime Database

In this section, we will first give a brief introduction to the two types of DBMSs we are going to use: Sybase and Objectstore. Then we talk about the crime database used for VPD system, including its schema and data set.

2.2.1 Relational DBMS and Sybase

Since the relational data model was proposed by E. F. Codd in the early 70s, relational DBMS (RDBMS) has been enjoying a great success[Silb91]. In the relational data model, data are organized into *tables*. Data records are represented as tuples in a table, which are rows of the table. The columns of a table represent attributes of data records. The user can only refer to a table. Any data record in the table can NOT be accessed directly. They can only be retrieved by queries. This is called the *associative access*, which is to look up a data record by the value of its attributes.

Sybase is one of the most powerful and efficient commercialized relational DBMSs. Sybase DBMS server version 4.9.2 is chosen as the relational DBMS platform to implement our VPD system. Sybase uses Transact-SQL, which is an extended form of standard SQL, as its query language.

Sybase uses a typical client/server architecture. The client program written in the host language (C in our VPD system) can send SQL queries using Sybase's DB-Library calls to a remote Sybase server through network. All the components in a SQL query, mainly including the projection predicates (select-clause), the table (s) to be used (from-clause) and the selection condition (where-clause), are sent to the server in one call. Then the client would wait for the final results from the server.

The server is in charge of the whole query processing. It interprets the SQL query and chooses a right processing plan to execute it. After finishing, it will return the final results back to the client through network. Sybase server has a very powerful query optimizer and is very good at utilizing existing indexes. This will be discussed in detail later. Normally, all the databases reside on disks attached to the Sybase server site CPU so as to achieve good performance. Fig. 2.2 illustrates Sybase's client/server architect.

2.2.2 Object-Oriented DBMS and Objectstore

In the past several years, *object-oriented* approach has been adopted successfully in programming languages and database developments[Kim90]. It has introduced a lot of new concepts into computer world, such as *object* and its *encapsulation*, *class* and

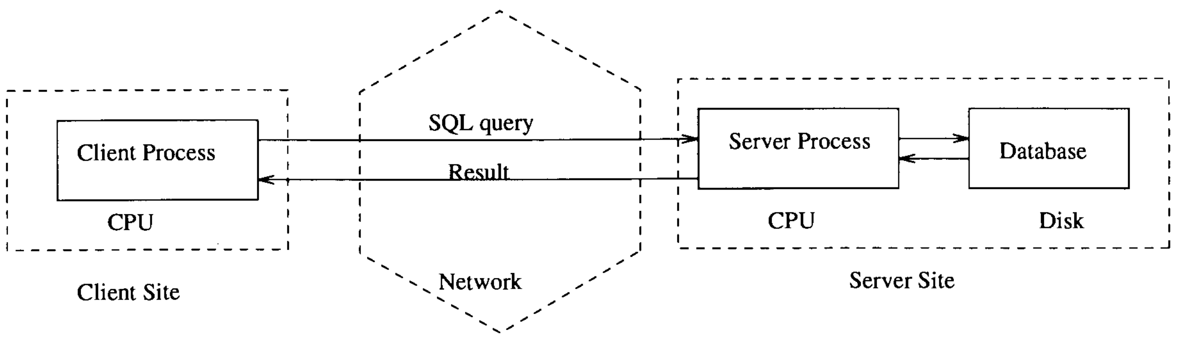


Figure 2.2: The Client/Server Architecture of Sybase

class hierarchy through *inheritance*. These Object-oriented data management methods gave birth to a new generation of DBMS, Object-oriented database management systems, OODBMS. OODBMS can more efficiently manage complex data objects, such as bitmaps, geometry data, texts, audio and video data, etc, than relational systems[Silb91]. Besides data *associative access*, OODBMS also supports data *navigational access*, which is to access data objects through pointers.

Currently, there are quite a few well-known object-oriented database management systems available, e.g. Gemstone by Servio Corp.[Butt91], Objectstore by Object Design Inc.[Lamb91], O2 by O2 Technology[Deux91], France and Postgres by University of California, Berkeley[Ston91]. Each of these OODBMSs has its own advantages and disadvantages. No one is perfect yet. Among them, we find out that Objectstore has quite a lot of attractive features and is most appropriate for our use. Therefore, we choose Objectstore as the OODBMS platform to develop our VPD system.

We use Objectstore version 2.0.1 by Object Design Inc. In our VPD system, we use C++ as the host language of Objectstore. Data actually stored in Objectstore databases are called *persistent data* by declaring *persistent*. Data only exist in memory (like the variables used in ordinary C++ program) are called *transient data*. Objectstore provides a tightly integrated language interface to the database persistent storage. This means that, C++ can define and handle persistent data objects in exactly the same way as transient data, except to put key word *persistent* and the database name in front of the variable declaration. Objects of any C++ data type can also be stored in Objectstore database as persistent data. All the C++ operations that can be applied to ordinary transient data can be applied upon persistent data.

Objectstore takes advantage of the native operating system's virtual memory management to manipulate persistent data with low overhead. This technique is called Objectstore's *Virtual Memory Mapping Architecture (VMMA).* It enables users to have direct and transparent access to persistent data with the speed of accessing transient data (it is also due to Objectstore's caching mechanism). These features give Objectstore a fairly high data access speed, especially for *navigational data access.*

Set-oriented data can be organized into a *collection*, a data structure provided by Objectstore. A *collection* contains a set of pointers to data objects of the same class. A collection has the functionality similar to a "table" in relational database but it contains the pointers to data objects instead of data themselves. The objects here can be of any class. The collection class in Objectstore supports a variety of member functions, such as inserting, removing, retrieving element pointers as well as set-theoretic operations like intersection and union.

Objectstore also supports associative queries, which are performed on collections. An Objectstore *query* is always applied on an Objectstore collection to return a sub-collection in which all the data objects qualify for the query conditions. The conditions are expressed in a string called the *query string.* The format of Objectstore query is:

Result_collection = Original_collection.**query** ("Query String")

query is a key word from Objectstore. The Original_collection is equivalent to the table name in SQL's From-clause. The Query String is a selection constraint, similar to the "where-clause" part of SQL. Some examples will be given later. Collections are declared as C++ variables and the **query** is executed as a C++ operation.

We can see that an Objectstore *query* has different meanings from a SQL *query.* An Objectstore query does not contain the "select-clause" part as contained in a SQL query. Its result is a collection of pointers. In order to get desired attribute values, we have to dereference each pointer in the result collection. This operation is called *retrieving* in this thesis. Therefore, one SQL query needs to be implemented by two steps in Objectstore, *query* and *retrieving.* Objectstore provides a function *foreach ()* to go through all the pointers in a collection. Objectstore also provides B-tree as index to enhance the query performance.

Objectstore is a client/server structured system, enabling distributed data access with concurrency control and recovery for transaction management. However, in contrast to Sybase, Objectstore executes all the queries in the client site. When a client process issues the first call to Objectstore server (normally the database open call), a file will be created as the *client cache* on the local disk of the client computer. A process called the *cache manager* will be created on the client site as well. The client cache is used to store all the persistent data needed by the client process, sent from the server. The cache manager is in charge of manipulating the client cache and coordinating with the Objectstore server, such as transferring data and keeping the data in the client cache consistent with the data in the databases in the server site.

When an Objectstore client process needs to access any persistent data, the cache manager will first check if all the data or collections required are in the client cache or not. If not, it will generate a page fault and inform the Objectstore server to fetch the necessary data from the database and transfer them to the client cache. After this, the query is executed exclusively in the client cache to generate the result for the client process. Fig. 2.3 illustrates this client/server architecture.

Note in Fig. 2.3, "Query" may refer to either data *associative access* or *navigational access.* In case of *navigational access*, the operation is dereferencing and the data objects to be de-referenced will be brought into client cache from database on the server site. For the *associative access*, the "Requested Data" are sent from the server to the client cache, containing the queried collection and some auxiliary data, such as the relevant indexes. Then the query is done at the client cache and a sub-collection is passed to the client process as the "Result".

Here, we can see that both the internal and external structures of Objectstore differ very much from those of Sybase:

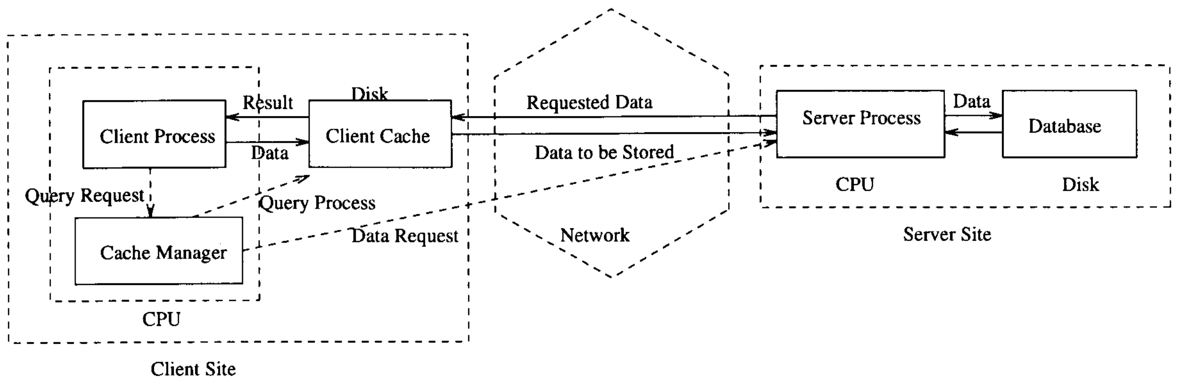


Figure 2.3: The Client/Server Architecture of Objectstore

- Internally, Objectstore allows each object to be accessed directly through *navigational access* but Sybase does not.
- Externally, Objectstore lets each client process do its own query at client site with the help of client cache. Its server is just a page server. Sybase does every operation at server site.

These differences let Objectstore have very high performance in processing non-record-oriented data, as demonstrated by [Zhou93] and [Wu92]. Later we will make use of Objectstore's these internal and external characteristics to optimize its associative query performance.

2.2.3 Data Types and Data Set

For the VPD project, we have a set of data records from the Vancouver Police Department. For every 911 complaint telephone call received in Vancouver City by the Police, a data record will be kept for this criminal incident. Each record contains the time, location, type and other relevant information of the complaint event. Following database naming convention, we call each type of these information an *attribute* of the record. The following is a list of the attributes in the crime record. The left column of the list is the abbreviate name of the attribute, the middle column is the more detailed description and the right column is the data type.

Abbreviation	Description	Type
compno	complaint ID	Integer
compdatetime	complaint date and time	String
cldatetime	close date and time	String
disptime	dispostition time	Integer
entret	time of 1st enroute	Integer
scenet	time of 1st on scene	Integer
stname	street name of the incident	String
stnumber	street number of the incident	Integer
beat	police beat	Integer
icompcod	initial complaint code	String
compcod	complaint code	String
pcode	priority	Integer
disposit	disposition	String
xcoord	x coordinate of the incident	Integer
ycoord	y coordinate of the incident	Integer

Table 2.1: The Original Data Attributes and Their Types

While the meanings of most attributes are quite clear, further explanations of some of them are still necessary. *compno* is an 8-digit integer unique ID assigned to each incident by the Police. *compdatetime* and *cldatetime* refer to the date and time when an complaint is reported and when the complaint case is closed by the Police, respectively. They are represented by character strings. For instance, when the date and time are "February 10, 1991 12:23:45", the string expression is "910210 12:23:45". The *disptime*, *entret* and *scenet* all refer to the times on the same day as the day of the complaint. These times are expressed by 6-digit integers. For example, "16:32:15" would be expressed by "163215". *stnumber* and *stname* combined together are the street address of the incident. For an address like "1234 ABC St", the *stnumber* would be "1234" and the *stname* is "ABC St". Complaint code means the type of the criminal incident. *icompcod* refers to the crime type being reported and *compcod* means the crime type determined by the Police, such as "THEFT", "ROBBERY", etc. *xcoord* and *ycoord* give the location of the crime. They are integer values measured by UTM, which is a world wide coordination system that can locate any spot in the world with X and Y coordinates. 1 unit of UTM means one meter. In the VPD system's crime database, the X coordinate is a 6-digit integer and the Y coordinate is a 7-digit integer.

For the query efficiency of the VPD system, 6 new attributes calculated or extracted from the original 15 attributes are added to the data set. They are:

Abbreviation	Description	Type
EA	value of the enumeration area	Integer
compdate	complaint date	Integer
comptime	complaint time	Integer
cldate	close date	Integer
cltime	close time	Integer
weekday	day of the week	Integer

Table 2.2: The Complemented Data Attributes and Their Types

EA means the value of *enumeration area.* The whole Vancouver city is divided into several hundred of enumeration areas for administration use, with each enumeration area having about 300 households. *compdate* and *comptime* are the date and time parts extracted from *compdatetime*, respectively. Both of them are represented by 6-digit (at most) integers. As an example, When *compdatetime* is "910305 06:34:05", the *compdate* is "910305" and the comptime is "63405". *cldate* and *cltime* are derived from *cldatetime* by the same principle. Certainly, the creation of these four attributes are replicate information and will bring some overhead to the space occupied by database. However, this will benefit and speed up the query processing, as some queries will query over just the date part or just the time part. *weekday* is the day of week when the complaint is reported. Thus it is calculated through *compdate.* From Sunday to Saturday, *weekday* is represented by numbers from "0" to "6".

Altogether these 21 attributes from Table 2.1 and Table 2.2 are put into 3 categories:

- **Temporal Attributes** The attributes carrying date or time information. Here we have *compdatetime*, *cldatetime*, *compdate*, *comptime*, *cldate*, *cltime*, *disptime*, *entret* and *scenet.* In our VPD project, most of the queries will be focused on the time when the complaint is reported, which is well related to *compdatetime, compdate and comptime.*
- **Spatial Attributes** *xcoord* and *ycoord* are the spatial attributes. They can locate the accurate spot where a criminal incident happens.
- **General Attributes** All the other attributes provide us with general information about the incident. They do not include much spatial or temporal information.

As we can see from above, these crime data records are just ordinary flat records with no complicated structures inside. All the attributes of them are eligible to be expressed by numbers or character strings.

The data set for running the VPD system, called VPDSET, has 148280 data records. They are all the records of complaints that happened in the first 6 months of 1991, which means that the attribute "compdatetime" of VPDSET ranges from "910101 00:00:00" to "910630 23:59:59". We use exactly the same names and data types of the 21 attributes to define the schema of a Sybase table called VPDTBL. In Objectstore, we still use these attributes as data members to define a class, *VPDDATA*, keeping the same names and types of the 21 attributes. Each crime record is created as an object of class VPDDATA. An Objectstore collection VPDCOL is created to contain pointers pointing to objects of class VPDDATA. After the definition, the VPDSET is loaded into Sybase table VPDTBL. In Objectstore, we store all the pointers pointing to objects in VPDSET to collection *VPDCOL.* We can see that both VPDTBL and VPDCOL contain exactly the same amount of same type of flat record data. All of our following experiments will be based on them.

The data records in VPDSET are ordered by the value of attribute *compdatetime*. So when these records are loaded into databases (either Sybase or Objectstore), they have reference locality with respect to attribute *compdatetime*. This means that any two data records, if the values of their "compdatetime" are close to each other, their physical locations in the databases will also be close. Clustered index is created upon *compdatetime* for both VPDTBL and VPDCOL. Nonclustered indexes are created on *comptime*, *xcoord* and *ycoord* for both VPDTBL and VPDCOL. These 4 indexed attributes will appear in our sample queries' constraints.

2.3 Query Parser

This section introduces the design of ESQL language and the Query Parser. The ESQL query language is designed for the front-end VPD system because some of the spatiotemporal queries are quite hard to be expressed in SQL. In order to let the system "understand" these ESQL queries and therefore execute them, a query parser has to be implemented.

2.3.1 Spatiotemporal Queries

Besides general query conditions, ESQL can also carry out queries for crimes which happened within a specified area in a certain period of time. The area may be regular, may be not regular. The time period sometimes may also get quite complicated to be expressed in SQL. Basically, the geometrical shapes that are eligible to be expressed by ESQL and processed by the VPD system can be a square, a circle or a buffer. A buffer here refers to the area within a certain distance to a street segment. For example, the area within 100m to street segment #200-#800 ABC St is a buffer.

More spatial shapes may be added to the list in the future, such as a random polygon, etc. The following types of time constraints have been implemented into the ESQL and the VPD system:

- **Time period:** from the starting date and time to the ending date and time, such as, from "Jan. 23 1991 02:34:23" to "Feb. 10 1991 20:31:00".

- **Time Frequency:** every day from a starting time to a ending time, such as, "every 10:00 - 13:34".
- **Weekday frequency:** every week from a starting day to a ending day, such as "every Wednesday" or "every Tuesday - Thursday".

Two or three of the above temporal conditions can also be combined together to do the query. For example, one can issue a query constraint meaning "From March 12, 1991 10:00:00 to May 20, 1991 23:12:00 every Fri. 15:30:00 - Sat. 5:30:00". This query will return the records of all the criminal incidents which happened after Friday afternoon 3:30pm, before Saturday morning 5:30am and in the period from March 12, 1991 10:00am to May 20, 1991 11:12pm.

Obviously, these application specific spatiotemporal queries are fairly hard to be expressed directly in SQL. This is why the ESQL is created. Our ESQL adopts SQL's basic query format, the *Select-From-Where clause.* Extra query expressions are added to the ESQL to handle spatiotemporal queries required by the VPD system. The basic framework of an ESQL query is:

```
SELECT selection_list
FROM database
WHERE spatial_conditions general_conditions
WHEN temporal_conditions
```

2.3.2 Syntax of ESQL

The ESQL is a case-independent non-procedural query language. Lower case letters and upper case letters are treated the same. The following is the BNF of the ESQL:

SELECT <Selection-List> [FROM <Database-List>] [WHERE <Where-Conditions>] [WHEN <When-Conditions>]

<Selection-List> ::= '*' | <Predicate-List>

<Predicate-List> ::= <Aggr-Predicate> [',' <Aggr-Predicate>] ...

<Aggr-Predicate> ::= <Predicate> | <Aggregate> ‘ (’ <Predicate> ‘)’
<Predicate> ::= <Attribute> | <Database> ‘.’ <Attribute>
<Aggregate> ::= COUNT | AVG | MAX | MIN | SUM
<Database-List> ::= <Database> [‘,’ <Database>] ...
<Where-Conditions> ::= { <Spatial-Conditions> , <General-Conditions>
}
<General-Conditions> ::= <Comparison-Expression> [{ AND | OR }
<Comparison-Expression>] ... | ‘ (’<General-Conditions> ‘)’
<Comparison-Expression> ::= <Attribute> <Comparison> { <STRING>
| <NUMBER> |<Attribute> | ‘”’ <STRING> ‘”’ }
<Spatial-Conditions> ::= CENTER ‘= (’ <NUMBER> ‘,’ <NUMBER>
‘)’ RADIUS ‘=’ <NUMBER>

| SQUARE ‘= (’ <NUMBER> ‘,’ <NUMBER> ‘,’ <NUMBER>
‘,’ <NUMBER> ‘)’
| STREET <STRING> RANGE <NUMBER> ‘-’ <NUMBER>
DISTANCE <NUMBER>

<When-Conditions> ::= { STARTING <Start-Time> [FOR <Duration>
] , EVERY <Period> }
<Start-Time> ::= ‘”’ <Date> [<Time>] ‘”’
<Date> ::= <Month> <Day> ‘,’ <Year> | <Day> ‘/’ <Month-Number>
‘/’ <Year>
<Month-Number> ::= <NUMBER>
<Duration> ::= { <NUMBER> YEARS, <NUMBER> MONTHS, <NUMBER>
WEEKS, <NUMBER> DAYS, <NUMBER> HOURS, <NUMBER> MIN-
UTES, <NUMBER> SECONDS }
<Period> ::= { <Time-Period>, <Day-Period> }
<Time-Period> ::= ‘”’ <Time> ‘” - ”’ <Time> ‘”’
<Day-Period> ::= <Weekday> ‘-’ <Weekday> | <Weekday> ‘”’ <Time>
‘”’ ‘-’ <Weekday> ‘”’ <Time> ‘”’

```
<Weekday> ::= <STRING>
<Time> ::= <Hour> ':' <Minute> [ ':' <Second> ]
<Month> ::= <STRING>
<Day> ::= <NUMBER>
<Year> ::= <NUMBER>
<Hour> ::= <NUMBER>
<Minute> ::= <NUMBER>
<Second> ::= <NUMBER>
<Comparison> ::= '=' | '<' | '>' | '>=' | '<=' | '!='
<Attribute> ::= <STRING>
<Database> ::= <STRING>
<STRING> ::= 'A'-'Z', 'a'-'z' ['A'-'Z', 'a'-'z','_', '0'-'9'] ...
<NUMBER> ::= '0'-'9' ...
```

The above BNF follows most of the basic expression rules for grammar's BNF design. Here are some more explanations:

- **Sharp Brackets** (< >) mean the item enclosed is a non-terminal identifier.
- **Braces** ({ }) mean that the at least one of the items enclosed must be chosen. When items are separated by (|), one and only one can be chosen. When separated by (,), one or more can be chosen.
- **Brackets** ([ ]) mean that none, one or more of the items enclosed can be chosen for use. When items are separated by (|), none or one of them can be chosen. When separated by (,), none, one or more of them can be chosen.
- **Single Quotes** (' ') enclose terminated symbols, which refer to symbols in ASCII table. Single quotes can be used with **dash** (-) to represent in ASCII table, all the symbols from the one before dash to the one after. For example, 'a' - 'z' means all the 26 lower case letters.
- **Ellipses (...)** mean the last unit can be repeated as many times as you want.

2.3.3 Examples

After introducing the syntax of the ESQL, let us look at some examples of how the ESQL queries really look like. "*vpddb*" is used for the database name of VPDSET.

> **Example1:** Find all the EA ids of crimes that happened from March 23, 1991 to May 10, 1991 from every Friday 8:30pm to Saturday 8:30am in the rectangle with lower-left corner being (483757 , 5455223) and upper-right corner being (487563 , 5458399) (which is UBC and its neighboring area).
>
> *SELECT ea*
> *FROM vpddb*
> *WHERE SQUARE = (483757 , 5455223, 487563 , 5458399)*
> *WHEN STARTING "March 23, 1991 00:00" FOR 1 MONTHS 17 DAYS*
> *EVERY FRIDAY "20:30"- SATURDAY "8:30"*

ESQL uses a starting time and a duration to express a time period. In Appendix A, some snapshots are provided to illustrate how this ESQL query *Example 1* is input from the VPD system's GUI by choosing various menu buttons and how its results are finally displayed on the GUI by coloring the map. They can give us a brief idea on the outlook of the GUI of the VPD system.

> **Example2:** Find all the locations of thefts happened within 500 meters to ABC St. #1200-#2000, for every week from Tuesday 16:30 to Wednesday 8:15.
>
> *SELECT xcoord, ycoord FROM vpddb*
> *WHERE STREET ABC St RANGE 1200 - 2000 DISTANCE 500 compcod = "theft"*
> *WHEN EVERY Tues "16:30" - Wed "8:15"*

Note that in the ESQL, the comparison of character strings is expressed the same as the comparison of numbers, such as "compcod = "theft"". Here we use *Example 2* to illustrate the efficiency and necessity of the ESQL. A street segment may not be a straight line, such as the ABC St shown in Fig. 2.4. The shadow area in Fig. 2.4 is the area to be queried by *Example 2.* Obviously, it would be very difficult and complicated for the user to use SQL or Objectstore's query string to express this spatial query constraint directly. However, it is quite straightforward to use the ESQL to express it. After this ESQL query is input from the GUI, the formidable task of executing it will be carried out by the VPD system's Query Parser and Query Processor.

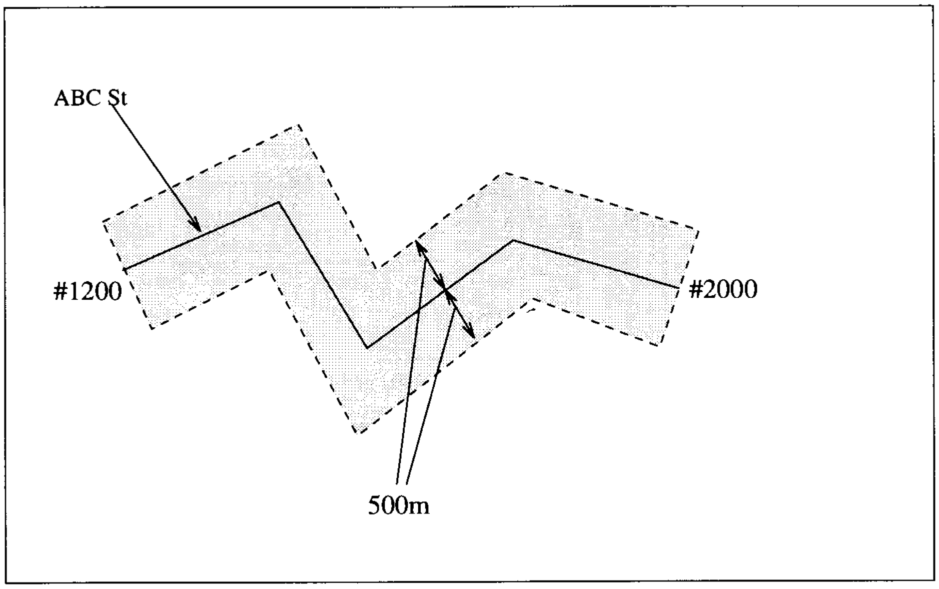


Figure 2.4: The Area To Be Queried by *Example 2*

2.3.4 Parsing of ESQL Queries

When an ESQL query is sent to the Query Parser to be parsed, all the information enclosed in the ESQL query will be recognized and saved in a structure, called the *parse tree.* This information basically includes the projection list, the database names and the selection constraints. The most complicated part is the selection constraints. In the VPD system, these constraints are classified into two categories, the *spatiotemporal constraints* and the *non-spatiotemporal constraints.* The *spatiotemporal constraints* refer to those ESQL-specific spatial and temporal query information, such as the constraints following "SQUARE", "CIRCLE", "STREET" and "STARTING", etc. The rest of the query would be similar to a normal SQL query, they are called the *non-spatiotemporal constraints.* The *spatiotemporal constraints* will be stored in special data structures, since these constraints occur very frequently and would be processed differently in the future. The *non-spatiotemporal constraints* are stored in an ordinary conjunction-disjunction tree (a binary tree whose internal nodes represent conjunctions or disjunctions and leaf nodes store comparison conditions). This tree and the spatiotemporal data structure make up the whole *parse tree.* The parse tree approach has been used in many systems[Oren92][Wu92]. Its detailed structure is not discussed here.

In the Query Parser, we provide the BNF of ESQL for the syntax and grammar checking. Along with the BNF, we will also code in the parser all the actions written in C++ to restore and save the query information in the *parse tree.* Once the parsing is done, the *parse tree* is sent to the Query Processor to start query processing. The ESQL parser is written in the format required by bison++. Then bison++ will translate the parser into C++ program. bison++ (a compiler-compiler) is the GNU C++ version of YACC.

2.4 Query Processor

The Query Processor has 3 versions. Here we introduce two of them. One utilizes SQL to execute ESQL queries on Sybase platform. The other one uses Objectstore collection query facility. We will also talk about the hardware configuration used for the Query Processor. First, we will see some ESQL sample queries which will be used to illustrate the basic mechanism of our Query Processor. They are also used for future performance tests.

2.4.1 Sample Queries

We use four sample ESQL queries to do most of our tests. All of them are spatiotemporal queries, featuring mainly on query spatial and temporal information, as desired by the VPD system. The 4 queries are named as QUERY1, QUERY2, QUERY3 and

QUERY4. All of the 4 queries will project over attribute *EA*, which is needed for counting the number of crimes that happened in each enumeration area. The table name for VPDSET will still be *vpddb*. The four queries are described in our ESQL language in the following.

The first query, QUERY1, queries over spatial attributes *xcoord* and *ycoord* and temporal attribute *compdatetime*. The number of data records in VPDSET that qualify for the selection criteria of QUERY1 is 4275.

QUERY1: Find the *EA* IDs of all the crimes that happened in a square whose lower-left corner is (483700, 5449400) and upper-right corner is (498450, 5460094) within the time period from 00:00am January 21, 1991 to 10:00am January 26, 1991.
SELECT ea
FROM vpddb
WHERE SQUARE = (483700, 5449400, 498450, 5460094) WHEN STARTING "Jan 21, 1991 00:00:00" FOR 5 DAYS 10 HOURS

QUERY2, QUERY3 and QUERY4 have the same query format. They all query over spatial attributes *xcoord* and *ycoord* and temporal attribute *comptime.*

QUERY2: Find the *EA* IDs of all the crimes that happened in a square whose lower-left corner is (488138, 5452243) and upper-right corner is (495764, 5457688) everyday from 10:00am to 11:00am. It returns 3073 data records, around 2% of the total records in VPDSET.
SELECT ea
FROM vpddb
WHERE SQUARE = (488138, 5452243, 495764, 5457688) WHEN EVERY "10:00:00" - "11:00:00"

QUERY3: Find the *EA* IDs of all the crimes that happened in a square whose lower-left corner is (486658, 5451608) and upper-right corner is (497071, 5457598) everyday from 1:15pm to 5:20pm. It returns 14731 data records, which is around 10% of the total of VPDSET.

SELECT ea
FROM vpddb
WHERE SQUARE = (486658, 5451608, 497071, 5457598) WHEN EVERY "13:15:00" - "17:20:00"

QUERY4: Find the *EA* IDs of all the crimes that happened in a square whose lower-left corner is (486227, 5452228) and upper-right corner is (495836, 5459609) everyday from 4:00pm to 9:15pm. It returns 28776 data records, which is around 20% of the total of VPDSET.
SELECT ea
FROM vpddb
WHERE SQUARE = (486227, 5452228, 495836, 5459609) WHEN EVERY "16:00:00" - "21:15:00"

Sample queries can be chosen in many ways. The reason why we use these four is that they are all spatiotemporal queries, covering the 4 most commonly used spatiotemporal attributes. For the sake of future performance analysis, since queries on clustered index *compdatetime* can be quite efficient, we only have QUERY1 to show this. Queries over other attributes need to use certain techniques to have better performances. That is why we have 3 queries QUERY2 - QUERY4 with the same format but different sizes of results to illustrate the potential improvement on them.

2.4.2 Two Versions of the Query Processor

Both the GUI part and the Query Parser part of the VPD system are independent from the DBMSs being used. Therefore their implementations are identical on either Sybase or Objectstore platform. Nevertheless, the Query Processor's task is to generate queries equivalent to ESQL queries and send them to DBMS. It relies on the underneath DBMS. Corresponding to Sybase table VPDTBL and Objectstore collection VPDCOL, we have two versions of Query Processor, named SYQP and OSQP, respectively, running over the two DBMSs. The reason why we have two query processors is not just for the VPD system, but also for future performance comparison and query optimization.

After receiving the *parse tree* from the Query Parser, the Query Processor would make use of the facilities provided by the DBMS to generate queries based on the information contained in the parse tree. These queries should carry out the functionality that the original ESQL queries are supposed achieve. The Query Processor then sends these queries to DBMS for execution. After the DBMS sends the execution results back to the Query Processor, it will forward the final results back to GUI.

The Sybase Query Processor SYQP's strategy is rather straightforward. It generates Sybase Transact-SQL queries from the *parse tree*, passes them to the Sybase server by Sybase DB-Library calls. After Sybase server finishes execution, SYQP will bind the returned result to a C language data structure and return the result back to GUI for display. Using the 4 sample queries as examples here, the SQL queries generated for our sample queries by SYQP are:

For **QUERY1:**
SELECT ea
FROM VPDTBL
WHERE xcoord >= 483700 AND xcoord <= 498450 AND ycoord >= 5449400 AND ycoord <= 5460004 AND compdatetime >= "910121 00:00:00" AND compdatetime <= "910126 10:00:00"

For **QUERY2**
SELECT ea
FROM VPDTBL
WHERE xcoord >= 488138 AND xcoord <= 495764 AND ycoord >= 5452243 AND ycoord <= 5457688 AND comptime >= 100000 AND comptime <= 110000

Since QUERY3 and QUERY4 have the same form as QUERY2, their SQL queries are generated with the same format as that of QUERY2. We can see that "*vpddb*" is translated to VPDTBL, which is the Sybase table name. All the spatial and temporal query expressions are rephrased in SQL format.

As Objectstore only provides a relatively lower level collection query facility, the OSQP's structure and mechanism are a little more complicated. After getting the parse tree, the OSQP takes two steps to accomplish the execution of an ESQL query. First an Objectstore query string is formed according to *parse tree.* Objectstore collection query is called upon the query string to get the result collection. This step is what we refer to as Objectstore *Query.* In the second step, the resulted collection will be gone through using Objectstore collection function *foreach* and the desired attribute (here is *EA*) is retrieved by dereferencing each pointer in the collection. The retrieved *EA* values will then be sent back to GUI. We call the second step *Retrieving.* Here are the Objectstore query strings generated by OSQP for our sample queries:

For **QUERY1:**
"xcoord >= 483700 && xcoord <= 498450 && ycoord >= 5449400 && ycoord <= 5460004 && strcmp (compdatetime, "910121 00:00:00") >= 0 && strcmp (compdatetime, "910126 10:00:00") <= 0"

For **QUERY2:**
"xcoord >= 488138 && xcoord <= 495764 && ycoord >= 5452243 && ycoord <= 5457688 && comptime >= 100000 && comptime <= 110000"

Query strings of QUERY3 and QUERY4 will be generated in the same way as for QUERY2. The Objectstore query string looks very similar to the where-clause part of the SQL query.

2.4.3 Hardware Configurations

In all of our performance tests, The Sybase server runs on a Sun Sparc 2 workstation with a 28.5 MIPS CPU and 32MB main memory. The Objectstore server runs on a SUN 4/300. Its speed is 16 MIPS and it has 8M main memory. These two servers can be accessed by client processes through network. The type of network used for all my experiments is Ethernet, which is a LAN (local-area network). Because a large amount of data will be transferred through network, the experimental results would differ greatly, should the experiments be carried out in another type of network, such as WAN (wide-area network).

There are two ways of choosing the client computers for the performance testing. One is to use the same site as the DBMS server, the other is to use a remote computer. It is actually a trade off to choose to use either of them. By using the server computer also as the client site, it can improve performance by reducing network usage since the database is also on the local disk of the computer. However, this will let the client process compete with the server process on resources such as CPU runtime and main memory. On the other hand, using a remote computer as the client processor can improve performance by avoiding this competition on computer resources but will increase network communication delay.

According to the argument in the last chapter, we know that the Sybase client computer plays an insignificant role in the query processing and the amount of data transferred through network is usually nominal. We decided to use a remote computer to run the client. The Sybase version of VPD front-end system is running on a SUN Sparc IPX workstation with a 28.5 MIPS CPU and 16 MB main memory.

For the Objectstore query, there will be a huge amount of data being passed through network, we first chose Objectstore server also to run the client process. The Objectstore client cache's size can also be set by the user. At first, its size is set to be its default value, 8MB.

After we set up our experimental environment, we can run the VPD system on both Sybase and Objectstore, record the run time of our 4 sample queries, then do performance comparisons between the two DBMSs.

From the performance point of view, the times taken by the 3 parts (DBMS execution time is combined into Query Processor running time) of the VPD system differ each other a lot. The GUI takes very little time for input and about 2-3 seconds for displaying the result. The query parser takes around 3-5 seconds to compile the query and generate the *parse tree.* Most importantly, the time taken by these two parts are nearly constant, independent of the DBMS platform and the form of the query. The major part of time consumed by VPD system is the query processing time. It varies a lot (from several seconds to more than one thousand seconds), depending on the DBMS platform, the form of the query, the number of returned data items and many other database related factors, which will be discussed later in this thesis. Thus, the following of the thesis will be focused on the performance of Query Processor and effective improvements.

Chapter 3

Query Optimizations for the VPD System

In this chapter, I first test and compare the performances of the VPD system query processing on two platforms: Relational DBMS, Sybase and Object-oriented DBMS, Objectstore. Then I analyze the factors that affect these performances, especially those on Objectstore. I propose two major categories of improvements to optimize the VPD query performance on Objectstore. Internally, I rearrange indexes in VPD-COL and sort the object pointers before retrieving them to enhance query efficiency. Externally, improvements are made to the VPD system's client site environment using Objectstore's client-processing-all feature and the "replicate database" approach to improve Objectstore's processing speed. As shown in the results, these optimization strategies have a tremendous impact on Objectstore performance and improve the VPD query speed on Objectstore dramatically. All the performance tests in this thesis are done at late night when the network traffic and impact from other users are minimized. All these tests exhibit fairly consistent speed from run to run. Throughout the thesis, once an improvement is made to the VPD system, it will be in effect for all the experiments after it. All the times in this thesis are measured in seconds.

3.1 Tests Comparisons between Sybase and Objectstore

In this chapter, all the tests are done using the four ESQL sample queries introduced in the last chapter. Table VPDTBL is used for tests on Sybase DBMS and collection VPDCOL is used for tests on Objectstore. The first set of tests is done on both Sybase and Objectstore. The system configurations are described in Section 2.5.3. The result is shown in Table 3.1.

Processing Time (sec.)		QUERY1	QUERY2	QUERY3	QUERY4
Sybase Query		5.0	45.7	56.6	60.7
Objectstore Query	Database Open	22.2	29.5	29.4	30.8
	Query	715.2	377.0	385.1	458.3
	Retrieving	22.4	72.3	167.4	915.1
	Database Close	2.9	1.1	1.4	1.2
	Total	769.7	479.9	583.3	1405.4

Table 3.1: Processing Time Comparison between the Sample Queries on Objectstore and Sybase.

In Table 3.1, we compare the processing times of the 4 sample queries on Sybase and Objectstore. For Sybase query processing, since the database open and close time is very insignificant (usually within 3% of the whole processing time), they are combined into the total processing time instead of being listed separately. Also for Sybase processing, the selection and projection are done in one step, so in the table we only have one row for the total Sybase query time. The Objectstore query processing consists of 4 major steps: Database Open, Querying, Retrieving and Database Close. They are listed separately in the table. The "Total" row refers to the whole processing time.

From Table 3.1, we can observe the following trends:

- For Objectstore queries, query and retrieving times count for the major part of the total processing time, yet the database open time is constantly fairly large too.

- Referring to QUERY2, QUERY3 and QUERY4, which are in the same query format, as the amount of returned results is increased, the processing times on both Sybase and Objectstore are increased. The increases are more dramatic for the retrieving times on Objectstore.
- Referring to QUERY1 and QUERY2, although QUERY1 returns more results than QUERY2 (4275 vs. 3073), QUERY1 takes much less time than QUERY2 for Sybase query and Objectstore retrieving. This is because the data set is clustered along attribute *compdatetime*. This will be explained in more detail in next section.
- Overall speaking, the Objectstore processing time is much longer than that of Sybase.

Recall the hardware configurations mentioned earlier, since the server computer for Objectstore is not as fast as that of Sybase, especially it has less main memory. We cannot just judge the performances of Objectstore and Sybase according to Table 3.1. In the condition that we cannot change the server computers, however, we will find out the factors affecting the performance and therefore improve it.

3.2 Improving Internal Query Processing Technique

We will start from improving the internal query processing efficiency on the Objectstore platform. Objectstore query processing is composed of two major steps: *query* and *retrieving*. As we can see from Table 3.1, they are also the most time-consuming steps. In the following sections, we will try to choose a better way to handle these two steps so as to reduce their overheads.

3.2.1 Improving Query Speed

If we want to improve the efficiency of query, we would have to first study the query procedure in detail. We will use the computation time complexity to evaluate a query procedure. As we know, indexes always play a very important role in query processing. In most databases, including Objectstore and Sybase, indexes are implemented as B-trees.

Our target is to improve the Objectstore query performance (when we say query we refer to only the selection procedure to get the result collection). Suppose we have a data record set D of which X and Y are two attributes. Indexes are created on these two attributes. Suppose the number of data records in set D is N. When the query is only over one attribute, such as "$Xa \leq X \leq Xb$" or "$Ya \leq Y \leq Yb$", using B-tree index, the query time complexity is O(logN). Without indexes, the query becomes linear search and its time complexity is O(N).

When querying over more than one key, things become more complicated. First let us look at the case in which a range query is over two attributes. The selection condition of our query Q is "$Xa \leq X \leq Xb$ && $Ya \leq Y \leq Yb$". In set D, there are N_x data records qualifying for the condition $Xa \leq X \leq Xb$ and N_y data records qualifying for the condition $Ya \leq Y \leq Yb$. In the following, we introduce 3 types of basic methods which can be used for doing range query Q.

- **Linear Search:** This method searches all the records in set D one by one without the assistance of indexes to get the result collection qualified for the condition. The time complexity is O(N).
- **Index-Search:** This method consists of two steps. The first step is to use the index on either X or Y to get the intermediate result collection C_x or C_y which satisfies the condition $Xa \leq X \leq Xb$ or $Ya \leq Y \leq Yb$, respectively. The time used is O(logN). In the second step, a linear search has to be executed on this intermediate result to get the final result which also satisfies the other selection condition. The time complexity for this step is O(N_x) for C_x or O(N_y) for C_y. The total time used for the whole procedure is $O(logN) + O(N_x)$ or $O(logN) + O(N_y)$, depending on whether the first step is done on X or Y,

respectively.

- **Index-Intersection:** This method also consists of two steps. The first step is to use the indexes on X and Y separately to get the intermediate result collections C_x and C_y such that C_x satisfies the condition $Xa \leq X \leq Xb$ and C_y qualifies for $Ya \leq Y \leq Yb$. In the second step, the two intermediate collections are intersected to generate the final result. The time complexity for the whole process is O(logN) + O($N_x * N_y$). If the two intermdiate collections can be sorted along a certain key before the intersection, then the intersection will take O(logN) + O($N_z log N_z$). N_z is the smaller one among N_x and N_y.

In most cases, N_x and N_y are both much smaller than N. So usually, the *Index-Search* method is the fastest query strategy. Using this method, by choosing the smaller one between N_x and N_y and using the corresponding attribute's index in the first step, we can further optimize the query processing.

When the number of attributes being queried gets larger (our sample queries used in practice all have 3 attributes in query), the time complexity of *Linear Search* remains to be O(logN). For Index-Search, the query time is no more than the time used for query over 2 attributes. If we have the ability to choose the optimistic index, then the processing time could even be decreased. However, for the *Index-Intersection* method, the more attributes and indexes are involved in query processing, the longer is the computation time. For queries over multiple attributes, if there is an index available for each attribute, the *Index-Intersection* method becomes the worst among the three methods.

As stated in the last chapter, for both our VPDTBL and VPDCOL, we have already created indexes on all the attributes which will appear in the query selection conditions. Virtually, each of the above 3 methods is eligible to be used.

Sybase[Syba91] has a very powerful query optimizer and is fairly efficient in utilizing indexes. It uses the *Index-Search* method to do the query. It can also find out the best index to use. For each index created for a Sybase table, a distribution of data records is built and maintained. The whole data set is divided into equal-sized pieces, called *steps*, each of which contains a certain number of records. Statistics co-exist with each index, recording the range of the indexed attribute value in each *step* and the number of data records in each *step*. When Sybase is given a range query, the Sybase server will consult the index statistics of each queried attribute and calculate the approximate number of records returned from the query over each index. Then it will choose the index which returns the least number of records. Using this index with the *Index-Search* algorithm would result in the best query performance. This guarantees that the Sybase query has a comparably good performance, as shown in Table 3.1.

We also investigate the way how Objectstore handles queries by using of indexes. After experimentations and discussions with Object Design Inc., we found out the principles that Objectstore uses to process a query:

- When there is no index available, *Linear Search* would be used.
- When there is only one index available for a query, *Index-Search* will be used on this index.
- When there are multiple indexes available, Objectstore will use the *Index-Intersection* method to get the result. If there are still some other attributes in the condition without index, the above result is only an intermediate one and it will be scanned to get the final result qualified for the entire condition.

For QUERY1 - QUERY4 in Table 3.1, each query involves 3 attributes in the selection criteria. Each attribute has index. In this case, Objectstore would generate 3 intermediate collections using 3 indexes and then intersects the 3 collections into the final one. This process, as mentioned before, is very time-consuming. This is why the "Objectstore Query" row in Table 3.1 takes such a long time.

In view of this Objectstore's drawback on query optimization, we decide to delete the indexes built on *xcoord* and *ycoord* in VPDCOL. Therefore, for QUERY1, there is only one index on *compdatetime* and for QUERY2, QUERY3 and QUERY4, only the *comptime*'s index is available. In this way, Objectstore will be forced to use *Index-Search* method, which is the fastest, to execute the query. After the indexes deletion, we repeat our tests with the other conditions remaining unchanged. Since Sybase does not have the index problem, we only did our tests on Objectstore and listed the results in Table 3.2.

Processing Time (sec.)	QUERY1	QUERY2	QUERY3	QUERY4
Database Open	22.1	28.5	25.7	22.3
Query	36.6	128.1	115.7	127.6
Retrieving	4.6	12.9	149.4	719.8
Database Close	2.8	4.8	1.5	2.6
Total	66.1	174.3	292.3	872.3

Table 3.2: The Query Processing Time after Index Re-arrangement on Objectstore

Comparing Table 3.2 with Table 3.1, we can get the following facts:

- The database open and close times have not changed significantly.
- The most changed row, as we predicted, is the query time. The deletion of some indexes forces Objectstore to use the more efficient *Index-Search* query strategy. This greatly increases the query speed. For QUERY2, QUERY3 and QUERY4, their speeds are increased by around 3 times compared to Table 3.1. For QUERY1, the increase is even much bigger. This is because the database is clustered on the attribute *compdatetime*. We will elaborate this point later.
- The data attribute retrieving times are also generally reduced for the four queries. This is also a result from clustering and will be explained later.

There are two types of indexes, *clustered* and *non-clustered*. An index is called a *clustered index* when the actual data of a table or collection are physically stored in the order of this index. This means that the data items' logical order with respect to the clustered index is the same as their physical order in the database. For example, two data items would reside side by side if their values on the clustered index are next to each other. For a table or collection, there can exist only one clustered index. When the clustered index is implemented by B-tree, the actual data items themselves can be used as the leaf level of the tree. On the contrary, *non-clustered index* means that the index order is independent of the physical order of actual data. When two data items have close non-clustered index values, they do not have to be located closely in database. The leaf nodes of the B-tree index contain pointers to real data.

The query time consumed by using the *Index-Search* range query method on clustered index and non-clustered index is calculated below. The first step is to generate the intermediate result, say R1, by index. In this step, more nodes (leaf level nodes) need to be visited for the non-clustered index approach. This is because for clustered index tree, the leaf level is the actual data. However, the non-clustered index tree's leaf level nodes contain pointers to real data. Both the pointers in the leaf nodes and the real data would have to be visited for generating R1.

The second step shows more significant timing differences. The intermediate result R1 will be scanned through according to the index order. When the index is clustered, R1 exhibits a locality of reference. For the same amount of data, they occupy the least possible number of pages since they reside close to each other. Furthermore, the data items will be fetched into memory and examined in the order of their physical location. Data in the same page, which is the unit for data transferring between memory and secondary storage, will be visited together. Once a page is discarded by memory, its data will not be visited again. However, for non-clustered index, R1 does not have locality of reference any more. The same amount of data may reside in many more pages. As data visiting order has nothing to do with their location, even data in the same page are visited separately. When the memory size is smaller than database size, which is the usual case, one page may have to be fetched into memory several times because the page may have been swapped out after the first visit and before the next one. In this case, a lot of page faults may happen and could even cause thrashing. Overall, in the second step of *Index-Search* for range query, using non-clustered index would cause many more I/Os than using clustered index. As we all know, I/O speed is much slower compared to CPU computation time. For Objectstore, more I/Os also mean more network communications between server and client.

From both steps of the *Index-Search* query method, we can conclude that using a clustered index would be much faster than using a non-clustered index. Clustered index is created on attribute *compdatetime* for both VPDTBL and VPDCOL. QUERY1, which uses this clustered index, costs much less time than the other 3 queries. This is clearly demonstrated in the Sybase Query time in Table 3.1 and Objectstore Query time in Table 3.2.

In the Objectstore *Retrieving* procedure, the collection resulted from *Query* is scanned again to do the projection on the desired attributes (Locality of reference resulted from index clustering will also take effect). Although there are more data records being retrieved for the results from QUERY1 than QUERY2, these data are clustered in physical storage and therefore ocupy less pages than the results from QUERY2. In Table 3.2, this causes the retrieving time for QUERY1 to be much smaller than that of QUERY2.

In the experiments in Table 3.1, Index-Intersection strategy is used for query. Only pointers to data objects need to be used for intersection and actual data objects are not brought into cache and main memory in query time. Therefore they have to be brought in at the retrieving time. For the experiments in Table 3.2, Index-Search method is used for query. Actual data objects are fetched into memory and client cache in query time already. Thus, data transferring would take less time in the retrieving procedure. This is why the Objectstore *Retrieving* times for the four queries in Table 3.2 are all shorter than those in Table 3.1.

The deletion of indexes brings some improvement to query processing. However we have to realize that this is not practical since there may exist all kinds of queries. Apparently, no combination of indexes can guarantee that there would be one and only one index available for any query so as to force Objectstore to use the *Index-Search* query method. To solve this problem, a multi-dimensional index tree will be introduced in the next chapter.

3.2.2 Improving Retrieving Speed

Inspired from the discussion in last section, we start thinking that the bad performance of Objectstore retrieving may very likely be related to too many I/O operations. Let us call the collection returned from Objectstore query collection C. The number of pages occupied by data in collection C is fixed. This would be the minimum number of pages to be fetched into main memory. However, we can try to visit the data items in collection C in an order such that the data located in the same page will be visited consecutively. In this case, once the CPU finishes reading a page, it will never need the data on this page again. This means one page will only have to be brought into main memory once. We can achieve this order by using Objectstore facility to order collection C along attribute *compdatetime*. But that will first involve attribute retrieval and will be very slow.

We insert a sorting procedure in the Query Processor between query and retrieving. This procedure will sort all the data pointers contained in collection C, using the values of the pointers as unsigned integers. We choose the Quicksort[Corm90] algorithm to do the sorting. In collection C, pointers pointing to data in the same page must have consecutive values. Dereferencing the sorted pointers in collection C can guarantee that the number of pages brought into memory will be the minimum possible number.

Keeping the indexes and other conditions the same, we sort the pointers in collection C before the *Retrieving.* Obviously, this change will not affect the times used for database open, query and database close. Our experiments also proved this. Thus, in Table 3.3 we will only show the sorting time and the total retrieving time for the 4 sample queries and omit the times used for other steps.

Processing Time (sec.)	QUERY1	QUERY2	QUERY3	QUERY4
Sorting	0.26	0.25	0.81	1.7
Retrieving Total	3.9	8.3	97.3	111.0

Table 3.3: The Sorting and Retrieving Time for Sample Queries

When we compare Table 3.3 with the "Retrieving" row in Table 3.2, the following points are noted:

- The sorting time complexity is N(logN), N is the number of pointers to be sorted. Since the sorting only uses the pointer values without dereferring the pointers, the sorting time accounts for a very insignificant part of the total retrieval time. It is negligible and will be omitted from tables later on.

- For QUERY1, there is little improvement for the retrieving speed because in Table 3.2, the retrieving is done on a collection resulted from clustered index query. For QUERY2, QUERY3, QUERY4, the sorting helps to speed up the retrieving. In particular when the retrieved data quantity is big, like for QUERY4, the retrieving speed is more than 6 times faster than the speed in Table 3.2. This further demonstrates that our previous analysis is correct.

Our work has been concentrating on improving the VPD system's internal query processing technique, mostly improving the query and retrieval methods. The results accomplished are quite satisfying. In the next section, we will try to find some environmental factors that may affect the VPD system's query processing performance on Objectstore platform.

3.3 Improving External Environmental Factors

3.3.1 Improving the Client Site Machine

For Objectstore, query processing is done at the client site. As the database is a fairly big one (50MB), main memory becomes an important factor for performance. If the memory is not large enough, page swapping would be very often and even thrashing may happen. This will slow down the processing speed. It seems that if we want the VPD system to have a better performance, it needs a reasonably more powerful client host.

Sybase's query processing is done at the server site. For sake of fairness, we switch to run the VPD system's Objectstore version on a computer with the same system configuration as the Sybase's DBMS server computer. This assures that both the Sybase and Objectstore's actual query processing is done by the same CPU and main memory configurations. As a reminder, since we move the client process away from the host for Objectstore server, the network communication overhead will increase. Thus the performance result would be a compromise between a faster computer + a bigger memory and a slower data transportation through network. In this experiment, the improvements on index and retrieving that we made previously will still be in use. The other experimental parameters remain untouched.

The results of QUERY1 - QUERY4's processing time after the client site is switched are shown in Table 3.4.

Processing Time (sec.)	QUERY1	QUERY2	QUERY3	QUERY4
Database Open	8.8	6.9	8.6	9.0
Query	12.4	40.2	81.0	97.8
Retrieving	0.6	0.3	51.7	69.7
Database Close	0.1	0.1	0.1	0.2
Total	21.9	47.5	141.4	176.7

Table 3.4: The Sample Queries Processing Time after Client Host is Changed

While we are analyzing the result in Table 3.4, we would also elaborate on how the four steps, i.e. *database open, query, retrieving and database close* are executed:

- **database open:** All the databases have to be opened before use. So normally they are the first Objectstore calls that an application process makes. For the first-so-ever database open call the process meets, a client cache will be created as well as a cache manager for the application process. They will be in use without being removed or re-created until the process ends, no matter how the databases and transactions are operated inside the process. Besides this, for each database open call, the cache manager will try to communicate with the Objectstore server to get the relevant information on that database, such as the database handler and schema, etc into the cache. After the application process is moved to a faster computer with bigger memory, the creation and running of the cache manager become faster. This brings the performance benefit as we can see from Table 3.4. The database open times are reduced to around 1/3 of those in Table 3.1 and Table 3.2.
- **database close:** Like closing a Unix file, mostly the work here is for the cache manager to inform the Objectstore server the closing of the database. Similar

to the improvement in database open, the database closing time drops to a very small value when the process runs on the faster machine. Since the database close time becomes so insignificant, in the future it will be combined into the total processing time.

- **query and retrieving:** For the Objectstore query and retrieval, all the database contents that are needed by these operations, such as indexes and actual data, will be brought into the client cache from the Objectstore database by the client cache manager and the Objectstore server. These data will then be processed solely in the client machine. As we said before, the change of client machine provides a faster CPU, a bigger memory and a longer network delay. From Table 3.4, the time used for query and retrieval for each single query is less than the corresponding one in Table 3.2 and Table 3.3. This is because in our case, the improvement of CPU speed and memory size of the remote machine overcomes the delay caused by network communication.
- **Total:** The speed of each step of every query has benefited from the client machine change, the total speed is also faster. However, the total processing time for each query is still more than the processing time for Sybase. We will continue to reduce it in the next section.

We also tested the Sybase performance after changing its client machine to another computer with different CPU and memory configurations, there is almost no impact. This demonstrates that all the Sybase queries are done in server site and their performances do not have much to do with client machine.

3.3.2 Replicate Database

From our previous analysis, we can see that most of the delay comes from data transferring process which delivers data through a path *database* -> *server site memory* -> *client cache* -> *client site memory.* Due to its slow CPU and small main memory, the Objectstore server computer becomes a bottleneck. Network communication delay is also an important factor. If we want to further improve the performance, the main concern is to reduce the time consumed in the first two steps of the data transferring path.

One way of doing this is to use the idea of *replicate database.* If we could set up a local database for our application, we do not have to always consult the server and we can save a lot of network communications. Fortunately, the "client site processing" characteristic from Objectstore makes the replicate database technology possible.

Every Objectstore application has a client cache, which can act as a replicate database under certain conditions. First, it has to have a certain size. It is better for the cache to be able to hold most of the data that the queries would need, such as the collections and the indexes on them, etc. Secondly, an Objectstore application would abandon its cache every time it finishes. Thus we need to keep the application process carrying Objectstore calls always alive to "protect" the client cache and the cache manager, even though we can still close all the databases and stop the transactions when they are not in use.

The above two conditions are also very practical for our application. As our Objectstore database VPDCOL is around 50 MB, we increased the size of the client cache from 8MB to 64MB, which is more than enough to hold the whole collection and all its indexes. This size is still quite affordable in terms of the hard disk size. For each single query, the process circle is *Open database - Start transaction - Execute query - Finish transaction - Close database.* In order to keep using the cache, we have to leave the main process running even after one query circle is finished, waiting for new queries to be input. This requirement wouldn't cause any problem for the VPD front-end system. The VPD system's main process starts from displaying the GUI and will keep running anyways, ready to accept and process new queries at any time. This process will keep VPD system's client cache and cache manager from being destroyed.

The first query of an application will have to fetch data from server database VPDCOL to client cache. Since all the queries in the VPD system use the same collection, VPDCOL, and indexes facilities built upon it, once all (or most of) the necessary data are brought in by the first query, they can serve all the subsequent queries without many cache misses. Thus for the subsequent queries, the client cache becomes the replicate database and all the data there are ready for use. The queries will be done locally. By using terms in [Gray91], the very first query of the application, which has to fill up the client cache with a replicate database, is called the *cold query.* The subsequent queries, which will occur on the existing client cache and cached database, are called *warm queries.* All the experiments we did before on Objectstore are cold queries.

After we start the VPD system, we can manage to let the first issued query fill up the client cache with most of the data from VPDCOL for future use. This query is called the *warm-up* query. The following is the *warm-up* query *WQUERY* we use:

```
SELECT ea
FROM vpddb
WHERE SQUARE = (0, 0, 8000000, 8000000) WHEN EVERY "0:00:00
- 23:59:59"
```

As 8,000,000 is far larger than the maximum X and Y coordinate values in VPDCOL, we can see that this query will need to query and retrieve the whole data set. By the end when it finishes, all the data in VPDCOL shall already be transferred to client cache from the server. Without stopping the VPD system, all the queries after WQUERY will be executed in the local cache and become warm-queries. Keeping all the improvements made before, we changed the client cache size to 64MB and test the warm queries on the replicate cached database. The results are shown in Table 3.5.

Processing Time (sec.)	QUERY1	QUERY2	QUERY3	QUERY4
Database Open	0.3	0.1	0.1	0.3
Query	3.7	10.7	27.7	43.3
Retrieving	0.6	0.6	3.7	4.6
Total	4.6	11.4	31.5	48.2

Table 3.5: The Sample Queries' Warm Processing Time after Using the Client Cache as Replicate Database

Let us first look at the improvements made from using warm query and changing the cache size by comparing the results with those in Table 3.4.

- **Database Open:** For the warm query database open, neither client cache nor cache manager needs to be created. Most of the database related information is already cached, e.g. the database schema. Thus the database open time becomes negligible.
- **Query and Retrieving:** Because the cache is big enough and holds the whole database before the query, the query and retrieving can be done almost locally in the cache with few cache misses and little data transportation. This improves the speed of these 4 sample queries, with more improvement for queries returning more data.
- **Total:** Comparing Table 3.5 with the Sybase performance in Table 3.1, it is seen that for all the four queries, the Objectstore performances in Table 3.5 are better than those of Sybase. This is because all the queries are executed in cached database on client site, like a local database. A lot of network communications are saved. For queries on Sybase, their results would still have to be sent back to client from server through network.

As before, we also tried to find a similar way to improve the Sybase performance. Since there does not exist a client cache in Sybase and the Sybase server does all the query processing, we can't implement the replicate database strategy for Sybase and the warm queries used on Sybase bring in little benefit either.

Here is a little reminder. Even after most of the interested data are cached in client site, for every Objectstore system call, such as database open and query, the cache manager still has to contact the Objectstore server to make sure that the current contents of the cache are most up-to-date and consistent with the database. In our experiments, only read accesses from a single user are involved. When there are multiple users and read/write operations, it is the server and cache managers' responsibility to keep the server database and all the cached databases update and consistent with each other.

3.4 Analysis

In the previous sections, we improved the performance of our application on Objectstore by making changes in two major areas: improving the Objectstore's internal query process mechanism and the VPD system's external environment. Throughout the series of improvements we made, our sample queries' processing speeds on Objectstore have been raised by up to 180 times and exceed the speeds produced by Sybase. This even happens while the Objectstore server machine is slower and has less memory than the Sybase server computer. The following is the overview of the improvements we have made in this chapter:

- **Internal process:** Due to the lack of query optimization technique on Objectstore system, we add some query optimization methods into our VPD system Query Processor.
  - *Index Rearrangement:* By deleting some extra indexes in the VPDCOL, we force Objectstore to switch to use the correct query mechanism, the *Index-Search* method, from the *Index-Intersection.* This improves the Objectstore query speed.
  - *Result Sorting:* By sorting the result collection according to the values of data pointers before the retrieval, we raise the result collection retrieving speed.
- **External environment** Our results also show that it is very important to realize the special features of Objectstore DBMS architecture.
  - *Client Site* Changing client process away from the Objectstore server site gives the VPD system a faster CPU and a larger main memory. This obviously overcomes the disadvantage brought by more network communications.
  - *client caching:* The enlargement of the client cache size and the using of warm queries allow the client cache to act as a replicate database for the VPD system. This provides a dramatic performance improvement.

These results demonstrate to us that besides the importance of the data structure, which has been emphasized by the literature, the strategy for executing associative queries and the DBMS architecture are also very important elements of a DBMS. As they are always crucial for a good query performance, it is very necessary for a good DBMS to adopt the correct strategy and for the application users to take full advantage of the DBMS features in order to achieve a satisfying result.

It should also be noted that all the improvements that have been made are quite practical and can be generally used on any applications, except that of the index change, which is only a temporarily solution. Currently there are two Objectstore system indexes on the attributes *compdatetime* and *comptime.* They work fine for our 4 sample queries since these two attributes do not appear in one query at the same time. However, it is possible that one query may contain both of them in the selection condition. In that case, the Objectstore will adopt the *Index-Intersection* query strategy and the query processing will be slowed down.

The solution for the index problem proposed by us is that we also delete the Objectstore index on *comptime* so that the clustered index on *compdatetime* will be the only system index left. This index is very efficient for queries containing *compdatetime* in selection constraint (shown by QUERY1). For other type of queries, user-defined indexes will be created and used to speed up query processing. These indexes include multi-key index, such as the $KDB^+ - tree$ that is to be proposed in detail in next chapter. How would the performance of these user-defined indexes be and how do we use them? This is what we are going to illustrate in the following chapters.

Chapter 4

A Multi-key Index: $KDB^+ - tree$

In the previous chapters, the VPD system: a system focusing on processing spatiotemporal queries, was introduced. The query performance of this system was also tested and a lot of improvements were made to enhance the speed of query processing on Objectstore. One of the problems is that Objectstore uses *Index-Intersection* query strategy, which is a rather slow one. This problem was solved temporarily in the last chapter by deleting some indexes. To overcome this drawback from Objectstore and to achieve maximum performance for spatiotemporal queries, in this chapter, we present the $KDB^+ - tree$ as an indexing mechanism for multi-key spatiotemporal queries.

4.1 Multi-key Index

As introduced before, our VPD crime database is composed of discrete point data. The schema of the database consists of 21 attributes, each attribute is either an integer or a string. There does not exist any complex structure within data objects, nor inter-object pointers exist. The queries are mostly spatiotemporal queries with constraints on X, Y coordinates and time, like our sample queries. A query which queries over 2 or more attributes is called a *multi-key* query and we call the attributes being queried over the *composite searching keys* (or *composite keys*) of the query. For example, our sample queries are all multi-key queries querying over 3 attributes. *xcoord*, *ycoord* and *compdatetime* are composite searching keys for QUERY1. *xcoord*, *ycoord* and *comptime* are composite searching keys for the other 3 sample queries.

As stated before, an Objectstore index is built on only one attribute and is effective only for the query over this very attribute. Even though the original collection may have several indexes, each time a query can just take advantage of only one of them. Then either *Index-Search* or *Index-Intersection* has to be used to finish the whole query. If we want to finish a multi-key query in only one round just by searching through an index, we need to build our own multi-key index on all the attributes that appear in this query. In our case, they are three-dimensional data: X, Y coordinates and time. As we all know, B-tree has become widely used and is the standard indexing method for one dimensional key. There have been substantial efforts for building indexes on multiple attributes. We classify the existent multi-dimensional indexes into 3 categories.

The first category of indexing mechanisms are only used for organizing discrete points. We have the *KD-tree* by Bentley[Bent75] in 1975, the *Conjugation-tree* introduced by Edelsbrunner and Welzl[Edel86] in 1986 and the *Two-Dimensional Orderings* introduced in [Same89] by Samet in 1989. The second category of indexes can deal with not only points, but also other geometric primitives such as polylines, polygons. They are the *Quadtree*[Same84], the *R-tree* defined by Guttman[Gutt84] in 1984, the *Grid File* given by Nievergelt et al[Niev84] in 1984 and the *Field-tree* introduced by Frank et al. in [Fran83] and [Fran89]. The third category of index structures are only suitable for polygons. The *Cell Tree* belongs to this category. It was designed by Gunther [Gunt88] [GB88] [Gunt90] in 1988.

Quite a few variations of the original index designs also exist. Different variations may have different advantages and disadvantages and may fit in different situations. Most of the indexing mechanisms proposed are only suitable for building indexes on two dimensional data while some others, such as KD-tree and Quadtree, can be used as indexes on more than 2 composite keys.

After studying the advantages and disadvantages of the above three categories of indexing structures, We present our own tree index, the $KDB^+ - tree$, for the use of the VPD system. $KDB^+ - tree$ is derived from Bently's KD-tree and Robinson's

KDB-tree. It also inherits some advantageous features from B-tree. It is designed to match the special features of queries in VPD system.

For most indexing tree structures, including B-tree, KD-tree, KDB-tree and our $KDB^+ - tree$, one node in the tree normally occupies a page, which is the basic unit for data transferring between main memory and secondary storage. As data I/O takes much longer time than CPU calculation, the number of pages that have to be visited in an index search becomes the key measurement for searching performance.

There are two broad classes of associative queries. One is called *range query*, or *sequential query*. The selection constraint of this kind of query covers a range instead of a point. The following is an example of a range query: "Find all the bus stations in the rectangle area of $100 < X < 200$ and $150 < Y < 300$". The other type of query is named as *random query* or find operation. In this thesis, it is also referred to as the *exact match query*. The selection constraint of this kind of query specifies discrete values as constraints. For example, "Find the information about the bus station on the spot X=100 and Y=200" is a random query. The queries in VPD system mainly feature multi-dimensional range query over point data.

The query performance proves to be very good for a balanced index tree with a lower-bound for the number of children that a node can have, such as the B-tree. Suppose an index tree of this kind is built for V data objects with all the pointers to data objects in the leaf level of the tree, Let d be the minimum children number of a node. The height of the tree is $log_d V$. The number of nodes (pages) a random query has to access is only the height of the tree, $log_d V$. For a range query, suppose M data items will be returned after the query, roughly O(M/d) tree nodes would be visited. If either the tree is not balanced or no such children number d can be guaranteed, the query performance would be much worse. Therefore, the two requirements, balanced tree and minimum children number, are the key factors for an index tree's success. Our $KDB^+ - tree$ also qualifies for these two conditions and has a very low searching overhead. It is built on multi-attribute keys. We start our discussion with the KD-tree, the origin of the $KDB^+ - tree$.

4.1.1 KD-tree

KD-tree was defined in 1975 by Bentley[Bent75]. It can be used for storing K-Dimensional points. In 1979, Bentley and Friedman[Bent79] gave a revised version of KD-tree. We take a 2D case to illustrate it. A KD-tree is a binary tree comprised of internal nodes and leaf nodes. Each leaf node of the tree contains a 2D data point. Each internal node contains a splitting point (which is chosen by a certain criteria and is not a real data point) and corresponds to a rectangle (We suppose the region of the 2D data set is a rectangle or the out-bounding rectangle of it). The root of the tree represents the whole region to the interest. For each node on level L of the KD-tree (assuming the root is on level 1), its corresponding region is divided into two parts by X coordinate if L is odd or by Y coordinate if L is even. A splitting point is chosen so that when its corresponding rectangle is divided into two by its X or Y coordinate, the number of data points in the two sub-regions are equal (or nearly equal).

Let us look at the example in Fig. 4.1. The root corresponds to the whole region. We first sort the 8 points in the region according to their X coordinates, then we pick up a point *a* so that its X coordinate is smaller than 4 points' X coordinates and is larger than the other four's. We use this point *a* as the splitting point and divide the region by its X coordinate. We get 4 points A, D, E, F in the left sub-region and 4 points B, C, G, H in the right sub-region. Using the same approach along Y axis, we get points *b* and *c* as splitting points for the left sub-region and right sub-region, respectively. Then we use the Y coordinates of points *b* and *c* to do the further dividing and so on, until in every region there is only one data point, which becomes a leaf node.

In this way, the KD-tree built is a binary tree. Also, it should be balanced or almost balanced (the difference between the longest and shortest path from the tree's root to a leaf is at most 1). One of the drawbacks of this KD-tree is that every node has at most two children. Not only will this increase the height of the tree, but also the node size will be much smaller than the page size. This will decrease the disk space usage and efficiency of secondary storage access. More pages will have to be accessed and more I/Os will be needed for query. Another disadvantage is that the

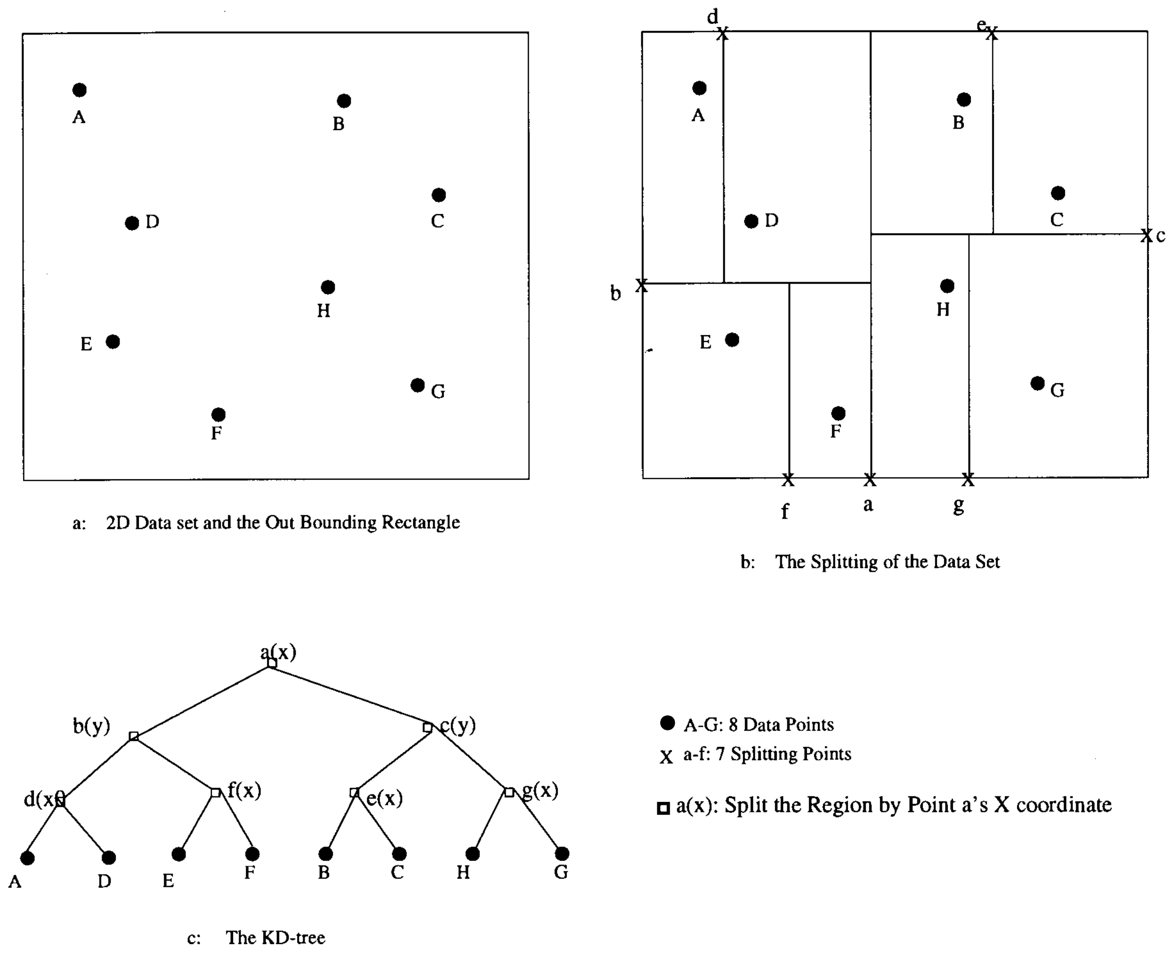

A
B
D
C
H
E
G
F
a: 2D Data set and the Out Bounding Rectangle
d
e
A
B
D
C
c
b
H
E
G
F
f
a
g
b: The Splitting of the Data Set
a(x)
b(y)
c(y)
d(x)
f(x)
e(x)
g(x)
A
D
E
F
B
C
H
G
c: The KD-tree
● A-G: 8 Data Points
x a-f: 7 Splitting Points
□ a(x): Split the Region by Point a's X coordinate

Figure 4.1: An Example of the KD-tree

creation of the tree is static. It would be very hard to dynamically insert or delete points while keeping the tree balanced.

4.1.2 KDB-tree

To cure the defects of the KD-tree, Robinson introduced the KDB-tree[Robi81] in 1981 by making some modifications to the KD-tree. Firstly, the KDB-tree is completely balanced. The path length from the root node to every leaf node is the same. Secondly, it is a multi-way tree. Its each node may have multiple children nodes instead of just two. Thirdly, it is dynamically created and maintained. We will use a KDB-tree built on 2-dimensional composite searching keys for the purpose of illustration. The same principle can be expanded to multi-dimensional KDB-tree indexes. Suppose the 2D keys are X and Y.

Each 2-dimensional KDB-tree node represents a rectangle, which is called the node's rectangle. Each node can have multiple children. The maximum number of children that any node can have is called the *bucket size* of the KDB-tree. No minimum children number is required. Each KDB-tree node is stored as a page in physical storage. Like the KD-tree, there are two types KDB-tree nodes, the *internal* nodes and the *leaf* nodes. Leaf nodes are also called *point nodes.* They are on the bottom level of the tree and a leaf node is composed of a collection of pointers to real data objects. For convenience, we also call each such pointer a child of the leaf node. Internal nodes are also called *region nodes.* The rectangle corresponding to an internal node is divided into several sub-rectangles. Each sub-rectangle is represented by a sub-node of that internal node.

Fig. 4.2 illustrates a 2-dimensional KDB-tree example.

In this 3-level KDB-tree, every node has up to 4 children nodes. The rectangle corresponding to each region is divided into smaller rectangles by X and Y coordinates alternatively. All the small regions resulted are DIRECT children of the split region page (the number of these small regions has to be equal or smaller than the predefined *bucket size*).

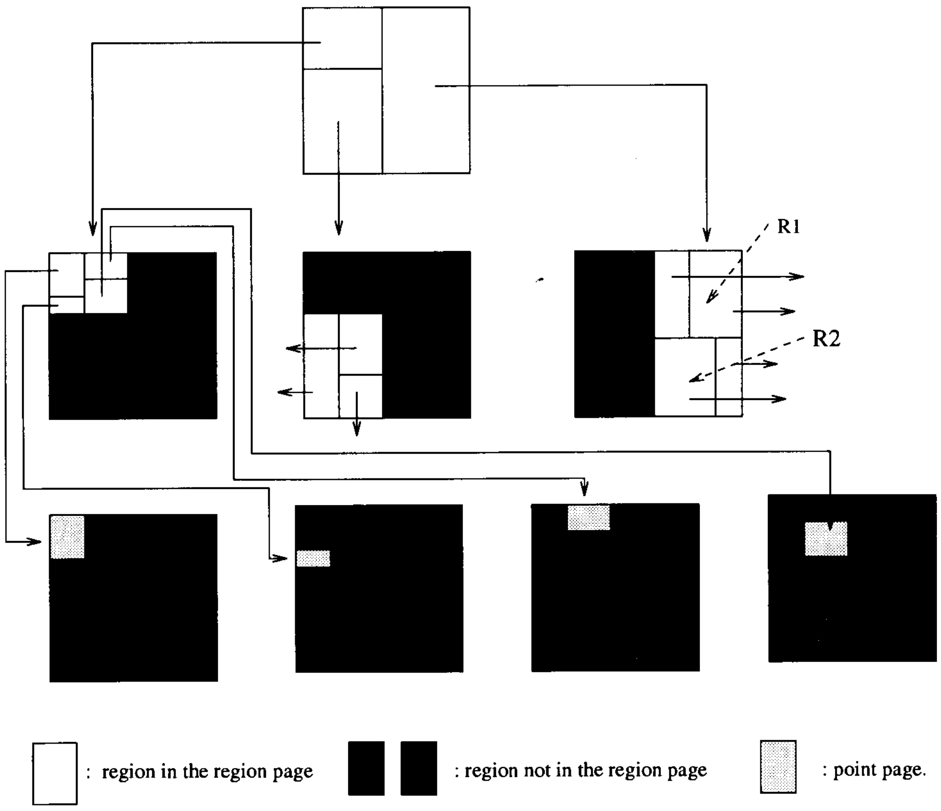


Figure 4.2: An Example of a 2-Dimensional KDB-tree

The major operations that can be applied on a KDB-tree are: query, insertion, deletion and splitting. These operations use the mechanism similar to the B-tree's[Come79]. For retrieving, inserting or deleting a data item given its composite keys (Xa, Ya), the method is to start from the root, going down the KDB-tree along a certain path such that the point (Xa, Ya) is inside the rectangles represented by the nodes on the path. At last a leaf node will be reached where the data item can be retrieved, deleted or the pointer to a new data item can be inserted.

As these operations deal with multiple dimensional keys, they have their own distinguishing specialty and are more complicated than operations on B-tree. Range query is to retrieve all the records in a specified rectangle R. The query mechanism is to start from the root of KDB-tree, go down the tree through paths on which all the nodes intersect with rectangle R, until leaf nodes are reached. In these leaf nodes, all the data which locate inside rectangle R can be retrieved.

After inserting a data item, if the number of data in a node exceeds the *bucket size*, splitting would have to take place. Without loss of generality, we assume the node is to be split by X coordinate. The leaf node's rectangle will be divided into two along X, with each new rectangle containing half of data in the old leaf node. In the KDB-tree, the old leaf node will be replaced by the two newly generated leaf nodes. The children number of their parent node will be increased by one. This is illustrated in Fig. 4.3a.

When the number of children of an internal node exceeds the *bucket size*, the node will also have to be split. Fig. 4.3b shows an example of splitting an internal node N. Suppose node N is split into two new nodes LeftN and RightN by a straight line L: X = X_0. The line is selected such that numbers of data items on both sides of the line are equal. For node N's each sub-node, SubN, it becomes a sub-node of LeftN if it is entirely to the left of L; it becomes a sub-node of RightN if it is entirely to the right of L. If line L goes through SubN's rectangle, SubN is also split by L into two new nodes. Its left part, node Left_SubN is a child of LeftN and its right part, node Right_SubN is a child of RightN. This splitting may go on all the way down to node N's descendent leaf nodes. This splitting procedure results in the major drawback of KDB-tree: there is no guarantee that SubN can also be split evenly by line L. It is even possible that one of Left_SubN and Right_SubN is completely empty and the other one has all the data items in SubN. In this case, there will be empty nodes in the KDB-tree and these empty nodes will still have empty sub-nodes and sub-sub-nodes as splitting of one node may cause the splitting of its descendent nodes.

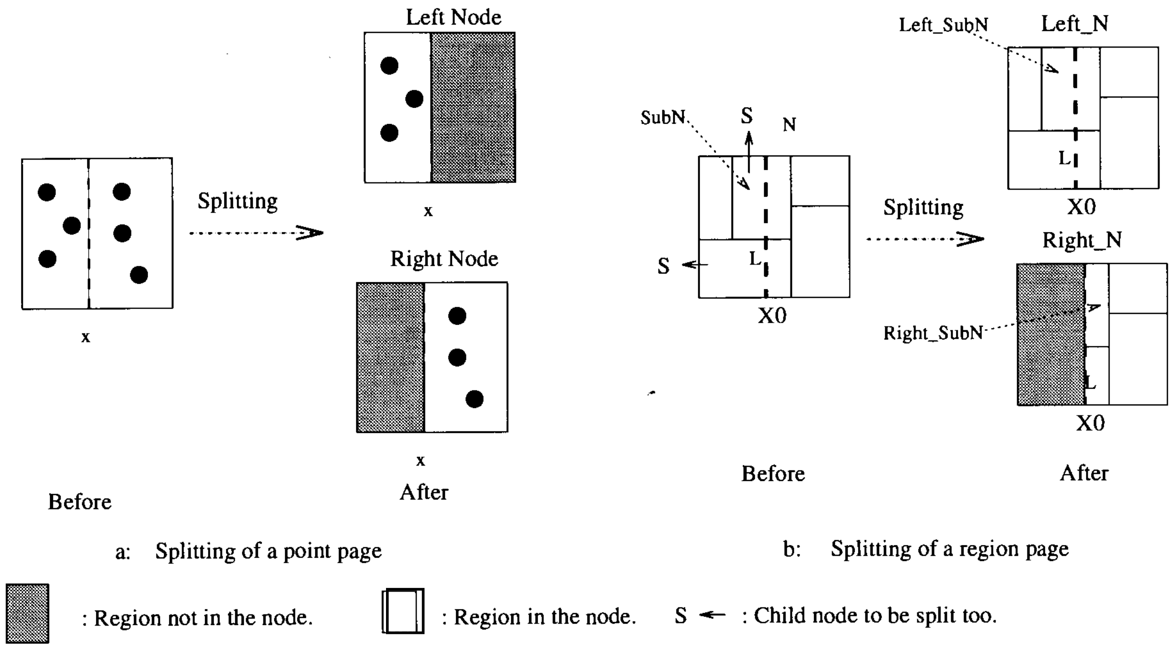


Figure 4.3: Examples of Splitting a KDB-tree Node

There are more problems with deletion. After a number of data items are deleted, certain leaf nodes and internal nodes could have fewer and fewer children. Although Robinson's KDB-tree allows this (even empty nodes are permitted), he still proposes a method to improve this situation by borrowing B-tree's approach, combining some low-storage neighboring nodes into one. However, in KDB-tree, not any two neighboring sibling nodes' rectangles can be combined into another rectangle, like R1 and R2 in Fig. 4.2. Nodes are called joinable if their rectangles can be combined into another rectangle. Therefore, sometimes more than two joinable nodes would have to be combined into a bigger one. In this case, the children number of this node may exceed the *bucket size.* Then splitting would have to be applied and new low-storage nodes may be generated.

Compared to KD-tree, Robinson's KDB-tree has certain advantages. It is dynamically created and maintained. It is balanced multi-way tree instead of binary tree. The major drawback of this data structure is that, unlike B-tree, it cannot guarantee the minimum number of children a node can have. For KDB-tree, since the children number d can be very small or even 0, there is no reasonable upper bound for the height of the tree. According to the analysis and calculation in Section 4.1, more pages will have to be accessed and more I/O operations have to be executed. These will result in a very poor performance for both random and range query operations on the KDB-tree. Also, the page occupancy will be low and lots of disk and memory space may be wasted.

Furthermore, sibling nodes can be different in each dimension of the composite keys. This would cost more calculation on each node to do exact point match or range intersection. In order to eliminate these drawbacks as well as to keep the advantages of the KDB-tree. I propose $KDB^+ - tree$ as a multi-key indexing mechanism.

4.2 $KDB^+ - tree$

Our $KDB^+ - tree$ is also a multi-way balanced tree. It eliminates the major defects of Robinson's KDB-tree. The nodes of a $KDB^+ - tree$ are constructed in such a way that all the sibling nodes only differ in one dimension and have the same range in the other dimensions. This makes the query much simpler since only one dimension needs to be checked for each node against the query range. Operation algorithms are all improved yet the biggest change comes from the revision of the splitting method. When splitting a $KDB^+ - tree$ node, all the data items in the node are redistributed and all the descendent nodes (to leaf nodes) are re-constructed in order to keep the children number of each node above a certain limit. None of the previous dividing boundaries are reused. This method eliminates the possibility of empty nodes and guarantees that the $KDB^+ - tree$ can keep a lower bound for its node's children number, besides the upper bound. Therefore, there is a lower bound for the tree's storage efficiency and an upper bound for the height of the tree. Query efficiency is on a higher level. A *static creation* algorithm is also proposed to create a $KDB^+ - tree$

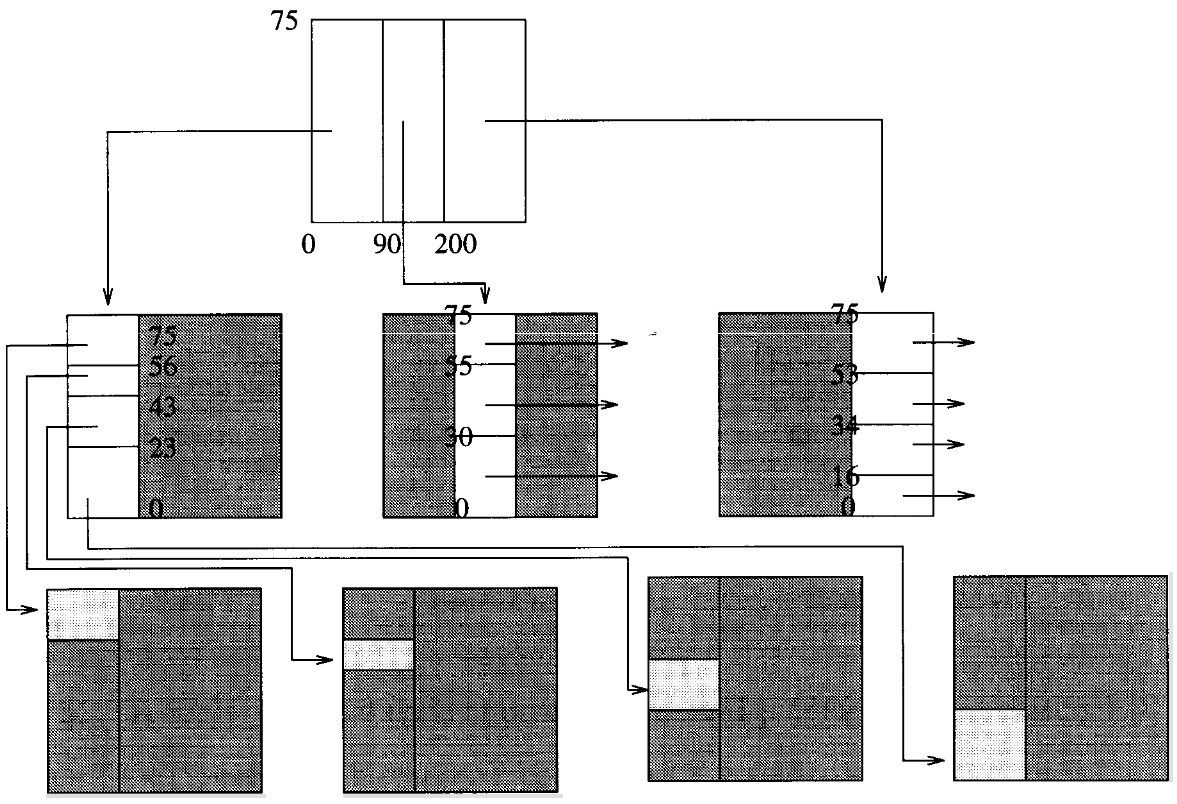
from static data. These changes in the structure of $KDB^+ - tree$ make it one step closer toward the B-tree. Compared to the KDB-tree, the $KDB^+ - tree$ is a lot more efficient, especially for query operation.

4.2.1 Structure

We still use X, Y as the 2-dimensional composite keys for the $KDB^+ - tree$ index. Like the KD-tree and KDB-tree introduced before, the nodes in the $KDB^+ - tree$ and their corresponding rectangular regions are also divided by X coordinate on the odd levels and Y coordinate on the even levels. Nevertheless, the most distinguishing feature of the $KDB^+ - tree$ structure is that each region node and its corresponding rectangle is divided into a certain number of rectangles just along one dimension, i.e. only along X axis or only along Y axis, depending on the level of the node. In fact, all the nodes on the same level of the tree are split along this dimension. This dimension is called the *designated dimension* of these nodes and this level. In this way, all the sub-nodes of a $KDB^+ - tree$ node only differ in one dimension (which is this node's designated dimension) yet have the same range in all other dimensions. Because the tree is a balanced tree, we can also define the level of the tree, we say that all the leaf nodes are at level 1 of the tree. All their parents are on level 2 and so on and so forth. Fig. 4.4 is an illustration of a 2-dimensional $KDB^+ - tree$.

As shown in Fig. 4.5a, when an internal node has n separation keys, these n keys divide the region along its designated dimension into n+1 sub-regions. It has n+1 pointers to these n+1 children nodes. In a $KDB^+ - tree$ of *order d*, except for the root node, each node can at most have 2d+1 pointers and must at least have d+1 pointers. This means a region node (except the root node) should have at least d keys and d+1 points and have at most 2d keys and 2d+1 data objects. The root node can have at most 2d keys and 2d+1 pointers too and must have at least one key and 2 pointers.

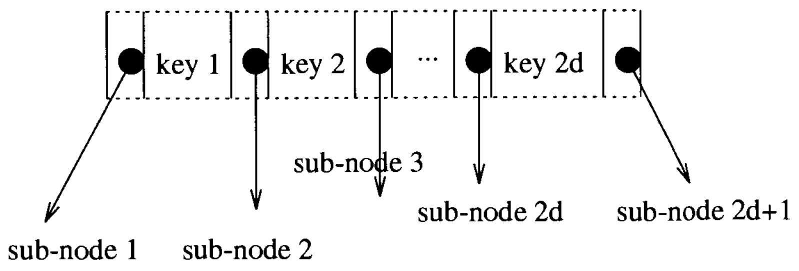
Besides the composite keys, the schema of data objects may also contain other attributes. Different data items may have same composite keys. Therefore, unlike the KDB-tree, the $KDB^+ - tree$ allows duplicate composite keys. In this case, the range



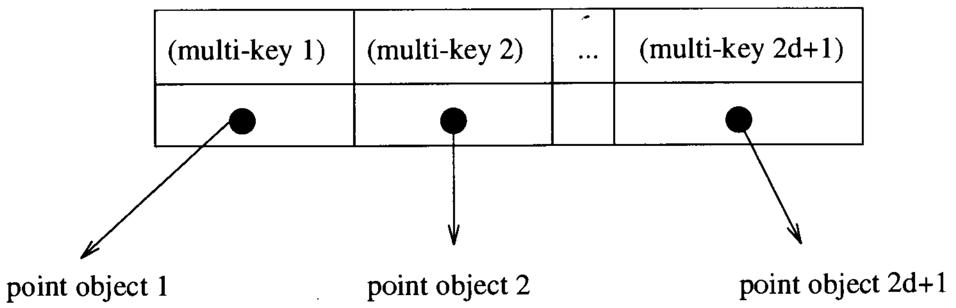
□ : Region in the region node ▨ : Region not in the region node. ▒ : point node.

Figure 4.4: An Example of a 2-dimensional $KDB^{+} - tree$

given by keys are always closed. For instance, the range between key_i and key_{i+1} should be $[key_i, key_{i+1}]$ instead of (key_i, key_{i+1}).



a. A KDB$^+$-tree internal node of order d with 2d+1 pointers to children nodes



b. A KDB$^+$-tree leaf node of order d with 2d+1 pointers to point objects

Figure 4.5: The Internal Structure of a $KDB^+ - tree$ Node

A leaf node in a $KDB^+ - tree$ contains pointers to data items and the composite keys of these data. Of course the number of data pointers and the number of composite keys are the same, shown in Fig. 4.5b (composite keys are called a multi-key there). In our example, the multi-key is composed of X and Y coordinates of the point. For a $KDB^+ - tree$ of order d, a leaf node should have at least d+1 keys and pointers and at most 2d+1 keys and pointers.

Thus, each $KDB^+ - tree$ node (except the root node) is at least 1/2 full. For the example shown in Fig. 4.4, its corresponding $KDB^+ - tree$'s internal structure is shown in Fig. 4.6.

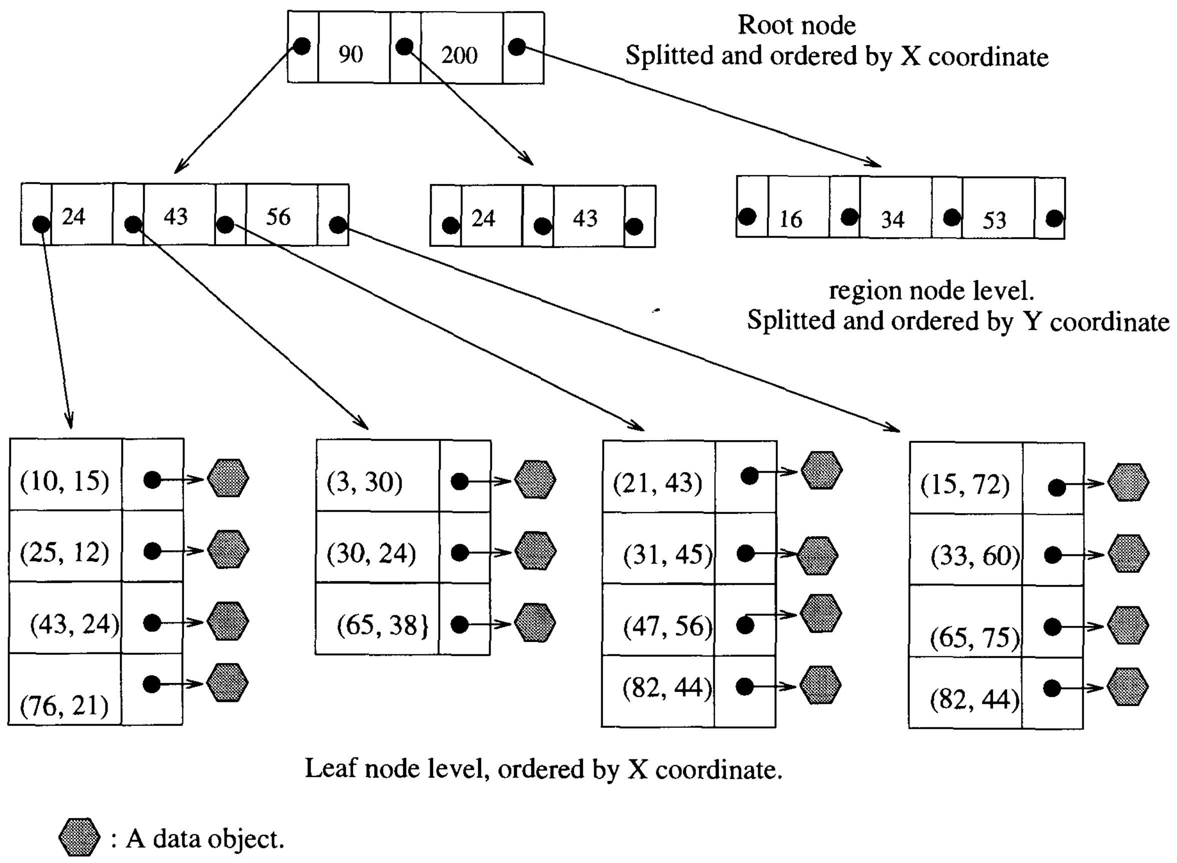


Figure 4.6: The Internal Structure of the $KDB^+ - tree$ Example in Fig. 4.4

4.2.2 Query

Due to the fact that all the sibling nodes differ only along their parent's designated dimension, the query algorithm, especially the range query of the $KDB^+ - tree$ is more efficient than that of the KDB-tree. Using Fig. 4.5 as an illstration, a range query algorithm for a 2-dimensional $KDB^+ - tree$ is described by procedure *RangeQuery (CurrentNode, qualify-info)*. *CurrentNode* can be any node in the $KDB^+ - tree$, *qualify-info* is an integer. The query is started by calling RangeQuery (root-node, 0). Following is the description of the algorithm:

GLOBAL :

A 2-dimensional $KDB^+ - tree$ with root node being *root-node.*

A rectangle query range on X and Y dimensions.

INPUT :

CurrentNode: $KDB^+ - tree$ node;

qualify-info: integer;

OUTPUT :

A set of pointers to data objects that fall in the query range.

BEGIN

- **Q1:** If CurrentNode is a leaf node, inspect all the multi-keys in CurrentNode and put all the qualified data pointers into query result. Terminate.
- **Q2:** Otherwise CurrentNode is an internal node. Suppose its designated dimension is X_des and the other dimension is expressed as X_und, the query range on X_des is [QR_MIN, QR_MAX]. There are n keys in CurrentNode. Each pair of consecutive keys in CurrentNode forms a range [key_{i-1}, key_i]. This range corresponds to a pointer to *sub-node*$_i$ (See Fig 4.5). For each such pair of keys, do the following (assume $-\infty$ as key_0 and $+\infty$ as key_{n+1}):
    - **Q2.1:** If range [key_{i-1}, key_i] does not intersection with [QR_MIN, QR_MAX], ignore it.

- **Q2.2:** If range $[key_{i-1}, key_i]$ is contained in the range of [QR_MIN, QR_MAX] ($key_{i-1} > QR\_MIN$ and $key_i < QR\_MAX$), this means *sub-node$_i$*'s rectangle qualifies for the query range on dimension X_des.
    - **Q2.2.1** If *qualify-info* shows that CurrentNode's rectangle already qualifies for dimension X_und of the query range, then *sub-node$_i$*'s rectangle qualifies for the whole query range. All the data items which are *sub-node$_i$*'s descendents are put into query result without further check.
    - **Q.2.2.2** Otherwise, store information "X_des qualifies" into *qualify-info* and execute QueryRange (sub-node$_i$, qualify-info)
- **Q2.3** If (key_{i-1}, key_i) intersects with [QR_MIN, QR_MAX], execute QueryRange (sub-node$_i$, qualify-info).

END

This algorithm is proposed for a 2-dimensional $KDB^+ - tree$. It can be easily expanded for use of multi-dimensional $KDB^+ - trees$. The basic principle behind this algorithm is to reduce calculation as much as possible. Following are the features present in the $KDB^+ - tree$ query mechanism but not in the KDB-tree one. First, for internal nodes, only range values on one dimension have to be checked. Secondly, once a node's rectangle is proved to be within the query region completely, all the data objects in this rectangle are sent to result automatically. This is what is done in step Q.2.2. These features result in the superiority of this algorithm over the query of KDB-tree.

A B-tree range query can be carried out by two *random queries*, but this is not feasible for $KDB^+ - tree$ range query. The reason is that here we have composite keys which are made up of 2 or more attributes. Therefore, no order can be defined among data objects. Since B-tree only uses one key as index, it can order all the data items into a chain. Therefore, it can turn a range query into two exact-match queries.

The exact match query using the $KDB^+ - tree$ is much simpler and is quite similar to that of the B-tree. It can also be done using the above algorithm by setting the query rectangle to be a point.

4.2.3 Static Creation

If we know a region and have all the data objects in this region, we can create a $KDB^+ - tree$ from root to bottom instead of inserting the objects one by one. This is called the *static creation* of the tree. In the *static creation*, we can do some pre-processing using the number of data items that will covered by the $KDB^+ - tree$ to decide how many sub-nodes an internal node will roughly have and how many data objects a leaf node will have so the tree can be created once and will not involve all the re-organization that the insertion causes. This way it is faster to create the tree and the user can have a better control over the shape of the tree.

For an order d $KDB^+ - tree$ with N($N \geq 1$) levels (called an N-level d-order tree), its root node shall have at least 2 sub-nodes and at most 2d+1 sub-nodes. Other nodes will have d+1 to 2d+1 sub-nodes. The *capacity* of this tree, which means the number of data objects that the tree can accommodate, is from $2(d+1)^{N-1}$ to $(2d+1)^N$. Therefore, when a $KDB^+ - tree$ is created as index for m data items, the order d and number of levels N of the tree can be chosen as long as they satisfy the condition $2(d+1)^{N-1} \leq m \leq (2d+1)^N$.

Definition 4.1 *A* strict $KDB^+ - tree$ *is a* $KDB^+ - tree$ *whose root also has at least* $d+1$ *sub-nodes.*

A subtree of a $KDB^+ - tree$ induced by an internal node other than the root node is a strict $KDB^+ - tree$. In most cases, *strict* $KDB^+ - trees$ are formed to become branches of another $KDB^+ - tree$. An N level *strict* $KDB^+ - tree$ with order d can be created for any m data items when $(d+1)^N \leq m \leq (2d+1)^N$.

Even after we can decide the order d and level N of a $KDB^+ - tree$, its shape can still vary. This is because the children number of any node can take any value from *d+1* to *2d+1*. Which value to pick is up to the user. For static creation, we recommend to distribute the data evenly throughout the tree, which means every node has the same number of children (or the difference of children numbers between different nodes is at most one). This way keeps the tree most balanced and searching will be efficient.

After the children number N_{root} for the root node and $N_{nonroot}$ for every other node are decided, the actual creation of the $KDB^+ - tree$ is straightforward. First, the root node of the tree is created. The whole data set is sorted along the root node's designated dimension. Then it is divided into N_{root} equal parts, with each part going to form a sub-tree of the root node. The separation key values and the pointers to these sub-trees are stored into the root node of the $KDB^+ - tree$. Using the same method and children number $N_{nonroot}$, all the root node's sub-nodes and all their descendent nodes can be created recursively. Down to the leaf level, leaf nodes are not split anymore. Composite keys and corresponding pointers to data objects are stored in leaf nodes.

Suppose the total number of data objects is V. For each level of the $KDB^+ - tree$, the V objects will have to be sorted along the level's designated dimension. The computation complexity for doing this is O(VlogV). The number of levels of the tree is O(logV). Therefore the complexity for the $KDB^+ - tree$ *static creation* is $O(V(logV)^2)$.

The static creation of the $KDB^+ - tree$ is most suitable when all or most of the data objects are ready for use before the creation of the $KDB^+ - tree$. Compared to online insertion of the data objects, the static creation can greatly reduce the tree creation time and keep the shape of the tree in control.

4.2.4 Splitting

For a $KDB^+ - tree$ with order d, if the number of children of a node exceeds the *bucket size* of the node, which is *2d+1*, the node needs to be split. Suppose in such a $KDB^+ - tree$ T, node A is the node to be split. Node A's rectangle is region R and let the set of all the objects contained in region R be S. Without loss of generality, suppose node A and its sibling nodes differ on dimension X. Following is the algorithm to split node A.

INPUT :

A $KDB^+ - tree$ T with order d. Overflowing node A in tree T.

OUTPUT :

A new $KDB^+ - tree$ T' after the splitting.

BEGIN

Use an appropriate value x_0 in dimension X to split region R into two new regions *R1* and *R2* and at the same time split S into two sub-sets *S1* and *S2*, such that the sizes of *S1* and *S2* differentiate from each other by less than 2.

Case I: If node A is a leaf node, create two new leaf nodes *A1* and *A2* corresponding to regions *R1*, *R2* to hold data sets *S1*, *S2* respectively. Replace the old node A in the tree with the two newly generated nodes *A1* and *A2*. This procedure is quite similar to that of the KDB-tree.

Case II: If node A is an internal node, suppose node A is on level N of the $KDB^+ - tree$. Based on regions *R1* and *R2*, use the *static creation* technique introduced before to create two N-level d-order *strict* $KDB^+ - trees$ *T1* and *T2* with root nodes being named as *left_node* and *right_node*. Replace node A with these two nodes so that the two new trees *T1* and *T2* serve as two sub-trees in tree T. The $KDB^+ - tree$ T is still balanced after the splitting. Let us prove that it is feasible to build 2 N-level d-order sub-trees *T1* and *T2* out of node A's sub-tree.

END

Definition 4.2 *The* volume *of any $KDB^+ - tree$ node is the total number of data objects which fall in the node's region. The pointers to these objects are contained in the descendent leaf nodes of this node. The volume of its root node is also called the volume of the $KDB^+ - tree$.*

Definition 4.3 *A* sub-chain *of any $KDB^+ - tree$ node A refers to the path and all nodes in the path from node A to one of its descendent leaf nodes, including node A itself and the leaf node.*

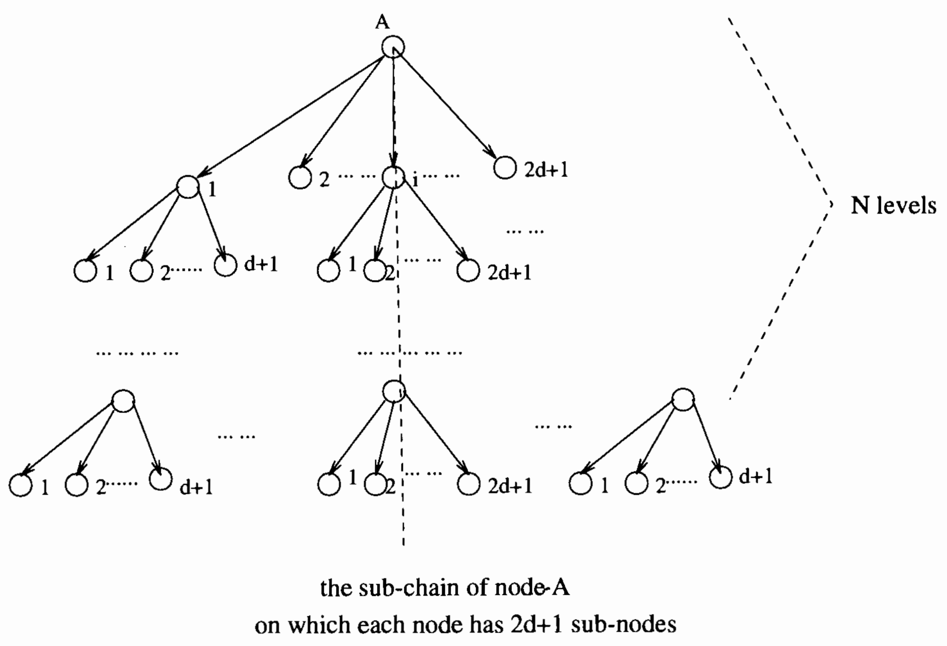


Figure 4.7: The Sub-tree of Node A when It is Going to be split

Let us see the volume of a $KDB^+ - tree$ node A when it needs to be split. Besides the extra data objects to be added, all the nodes in one sub-chain of node A must be full, i.e. they all have 2d+1 sub-nodes or data objects. This is the reason for splitting. The rest descendent nodes of A will have at least d+1 sub-nodes, as illustrated in Fig. 4.7. By calculation, we get that the volume of the to-be-split node A will be at least $2d(d+1)^{N-1} + 2d(d+1)^{N-2} + ... + 2d(d+1) + (2d+1) + 1$. We have the following Equation:

Equation 4.1 $2d(d+1)^{N-1} + 2d(d+1)^{N-2} + ... + 2d(d+1) + (2d+1) + 1 = 2(d+1)^N$, *$d$ and N are integers, $d > 0$ and $N > 0$.*

Proof:

$$\begin{aligned}
& 2d(d+1)^{N-1} + 2d(d+1)^{N-2} + ... + 2d(d+1) + (2d+1) + 1 \\
= & \ 2d[(d+1)^{N-1} + (d+1)^{N-2} + ... + (d+1) + 1] + 2 \\
= & \ 2d[(d+1)^N - 1]/[(d+1) - 1] + 2
\end{aligned}$$

$$= 2d[(d+1)^N - 1]/d + 2$$
$$= 2(d+1)^N \qquad \square$$

According to Equation 4.1, when node A is split evenly into *left_node* and *right_node*, the volume of each of the two resulted nodes will be at least $(d+1)^N$ and obviously, no more than $(2d+1)^N$. Therefore, the data objects contained by each of the *left_node* and *right_node* can form two new strict N-level d-order $KDB^+ - trees$ *T1* and *T2*. *T1* and *T2* will serve as sub-trees of the original $KDB^+ - trees$ T. Thus, after the node A is split and replaced by *left_node* and *right_node*, the tree T is still balanced. The splitting of a node can be shown in Fig. 4.8.

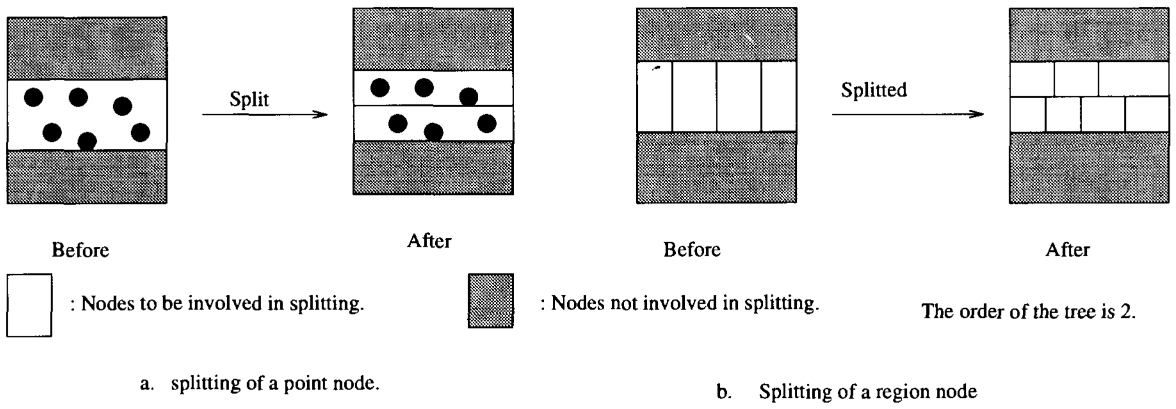

Split
Before
After
Splitted
Before
After
: Nodes to be involved in splitting.
: Nodes not involved in splitting.
The order of the tree is 2.
a. splitting of a point node.
b. Splitting of a region node

Figure 4.8: The Splitting of a $KDB^+ - tree$ Node

Let us now compare this splitting procedure with that of the KDB-tree. For the $KDB^+ - tree$ splitting, the split node and the sub-tree underneath it are going to be discarded and replaced by two new nodes and two brand new sub-trees. All the data objects in the split area are re-distributed and re-organized into new nodes in order to keep data objects scattered evenly. But in KDB-tree, all the original boundaries of descendent nodes of the split node will be kept for use. The improvement of the splitting procedure gives our $KDB^+ - tree$ a big advantage. The minimum number of children for any node can always be kept above the order of tree. The height of the tree will therefore have an upper bound of $1 + log_{d+1}(V/2)$, where d is the order of tree and V is the total number of data objects. On the contrary, because of the splitting strategy adopted by KDB-tree, its nodes could have very few sub-nodes or be completely empty. No upper-bound for height of the tree can be given.

When the volume of the split node is V, the splitting procedure is in fact *static creation* of two strict $KDB^+ - trees$, each of which has volume $V/2$. Therefore the computation complexity of the splitting algorithm is $O(V(logV)^2)$. In some cases, a $KDB^+ - tree$ node's splitting takes longer time than a KDB-tree (For example, when the dividing line of KDB-tree node A does not go through any of A's sub-nodes). However, when index is used, queries over the index are supposed to occur much more frequently than updating of the index. Improving query efficiency should have the highest priority. This is why we choose to adopt the $KDB^+ - tree$ approach. Since we know the splitting takes time, in the insertion of the $KDB^+ - tree$ we will try to use some special techniques to reduce possible splitting and the number of data objects involved in each splitting.

4.2.5 Insertion

First let us define the concept of *concatenation and redistribution.* On level N of a d-order $KDB^+ - tree$ T, for m consecutive sibling nodes A_1, A_2, ... , A_m whose ranges differ only on one dimension, say X, the following process is called the *concatenation and redistribution* to these m nodes:

As these m nodes' regions are all next to one another, we concatenate them into one region R, gather all the data objects residing in region R to be set S. Then along X, *m-1* values x_1, x_2, ... , x_{m-1} are selected to divide region R into m regions and set S is divided into corresponding m sub-sets so that the number of data objects in each region varies by no more than 1. Based on the m new regions, we can use the *static creation* technique to create m N-level d-order *strict* $KDB^+ - trees$ T_1, T_2, ... , T_m with root nodes being A'_1, A'_2, ... ,A'_m. In tree T, replace A_1, A_2, ... , A_m with nodes A'_1, A'_2, ... ,A'_m so the m new trees T_1, T_2, ... , T_m serve as m sub-trees in tree T. The $KDB^+ - tree$ T is balanced after the above *concatenation and redistribution* operation. When the total volume of all the nodes involved in *concatenation and redistribution* is V, the computation complexity is $O(V(logV)^2)$ since the main operation is still *static* *creation.*

Definition 4.4 *A node on level N of a d-order* KDB^+ *– tree is said to be* strictly full *when its volume is* $(2d+1)^N$.

In the insertion and deletion algorithms, we are going to use the methods of *splitting, concatenation and redistribution* discussed before. Once splitting is used, in KDB^+ *– tree T*, the two new nodes *A1* and *A1* and their sub-trees will replace the split node *A* and it is original sub-tree. To do this, in node *A*'s parent node *B*, the pointer to node A will first be deleted. Then the pointers to nodes *A1* and *A2* and the splitting key value are inserted into node *B*. Once *concatenation and redistribution* is used, in KDB^+ *– tree T*, the *m* new nodes *A1', A2' ... Am'* and their sub-trees will replace the *m* concatenated nodes *A1, A2 ... Am* and their original sub-trees. To do this, in node *A1*'s parent node *B*, the *m* pointers to nodes *A1, A2 ... Am* and *m-1* dividing keys between them will first be deleted. Then the *m* pointers to nodes *A1', A2' ... Am'* and the *m-1* new dividing keys between them are inserted into node *B*. As a convetion, this replacing procedure will not be explained again in detail in the following algorithms. The following is the algorithm for inserting a data object whose k-dimensional composite key is $(x_1, x_2, \ldots, x_k)$ into a KDB^+ *– tree*:

INPUT :

A k-dimensional KDB^+ *– tree* T with order d.

A data object to be inserted with composite key being $(x_1, x_2, \ldots, x_k)$.

OUTPUT :

A new KDB^+ *– tree* T' after the insertion.

BEGIN

- **I1:** Do an exact match on $(x_1, x_2, \ldots, x_k)$ and find the leaf node *current_node* where the new data object is supposed to reside. Insert pointer to this data object and its composite key into *current_node.* If *current_node* does not overflow, terminate. Otherwise go to step I2.

- **I2:** If the *current_node* is the root node, split *current_node* into two new nodes and create a new root node as the root of the newly created two sub-trees. Terminate. Otherwise go to I3.
- **I3:** Let *parent_node* be *current_node*'s parent node. If *parent_node* has less than 2d+1 sub-nodes, split *current_node* and do the replacing. Terminate. Otherwise go to I4.
- **I4:** If *parent_node* is not *strictly full*, find *current_node*'s closest sibling node A which is not *strictly full*. Do *concatenation and redistribution* to *current_node* and A and all their sibling nodes between them. Do the replacing in *parent_node.* Terminate. Otherwise, let *current_node* be *parent_node* and go to I2.

END

The major characteristics of this insertion algorithm is that it makes use of the *concatenation and redistribution* technique, which is not used by insertion algorithm of either the B-tree or the KDB-tree. Its use can reduce the number of data objects involved in most splitting procedures and can therefore improve the insertion speed. This technique guarantees that a single node is split only when it is *strictly full*, which means that its volume is $(2d+1)^N$. As a matter of fact, the use of *concatenation and redistribution* is optional. Its execution can be omitted from step I4 without causing any problem. However, the omission will result in that more data objects will be involved in the splitting which is used to replace the *concatenation and redistribution.*

Another feature of this insertion algorithm is that once splitting is decided to take place, it will not be spread to upper level nodes. In other words, we do not do splitting until we are sure that splitting the current node can solve the problem thoroughly and this splitting will not cause overflow to upper level nodes. It is guaranteed that no operation will be wasted.

Suppose the volume of the $KDB^+ - tree$ is V when insertion happens. O(logV) operations are needed to locate the leaf node where the data item should be inserted. If no overflow occurs, that is the computation complexity of the insertion. When overflow does occur, suppose V' data objects are involved in splitting or *concatenation and redistribution*, the complexity should be $O(V'(logV')^2)$. Therefore, the computation complexity of insert may vary from $O(logV)$ to $O(V(logV)^2)$, depending on which level the overflow occurs.

Now there are two ways to create a $KDB^+ - tree$. One is through the static creation, the other is through inserting all the data objects one by one. The static creation is faster and makes it easier to control the shape of the tree. However, it requires that all the data objects are available prior to the tree creation, which is not always possible. Using the insertion algorithm is more dynamic. These two methods can also be combined together for use. Data objects can still be inserted dynamically after the tree is built up by *static creation.*

4.2.6 Deletion

Opposite to the insertion procedure, the deletion of a data object may cause the volume of a leaf node to be less than the order d of the $KDB^+ - tree$. We say that such a node *underflows.* Certain actions have to be taken to keep up the children number for the $KDB^+ - tree$ node where deletion takes place.

Definition 4.5 *A node on level N of a d-order* $KDB^+ - tree$ *is said to be* strictly poor *when its volume is* $(d+1)^N$.

The following is the algorithm for deleting a data object with k-dimensional composite key $(x_1, x_2, \dots, x_k)$ from a $KDB^+ - tree$:

```
INPUT :
    A k-dimensional $KDB^+ - tree$ T with order d.
    A data object to be deleted with composite key being $(x_1, x_2, \dots, x_k)$.
OUTPUT :
    A new $KDB^+ - tree$ T' after the deletion.
BEGIN
```

- **D1:** Do an exact match query for $(x_1, x_2, \dots, x_k)$ and search for the leaf node where this data object resides. If the search is successful, name the leaf node as *current_node*. Delete the pointer of this data object and its composite key from *current_node*. If *current_node* does not underflow or *current_node* is the *root_node* of the tree, terminate. Otherwise go to D2. If the searching is not successful, generate error message and terminate.

- **D2:** Let *parent_node* be the parent of *current_node*. If *parent_node* has sub-nodes which are still not *strictly poor*, find *current_node*'s closest sibling node A which is not *strictly poor*. Do *concatenation and redistribution* to *current_node* and A and all their sibling nodes between them. Do the necessary replacing in *parent_node*, terminate. Otherwise go to D3.
- **D3:** If *parent_node* has more than d+1 sub-nodes or *parent_node* is the root node, combine *current_node*'s region with one of its sibling node A's region (A must be strictly poor). Suppose *current_node* is on level N, this new region contains $2(d+1)^N - 1$ objects. Use *static creation* to create a *strict* N-level d-order $KDB^+ - tree$ rooted at node P (according to Equation 4.1, this is feasible). Replace *current_node*, node A and their sub-trees with the newly created node P and its sub-tree. If the *parent_node* is the root and node P becomes its only child, delete the *parent_node* so node P becomes the root node of the $KDB^+ - tree$ and terminate. Otherwise let *current_node* be the *parent_node* and go to D2.

END

Using the insertion and deletion algorithms can dynamically maintain a $KDB^+ - tree$. Using the static creation can build up the tree in a shorter time. A better way to be used for creating and maintaining a $KDB^+ - tree$ is to combine them together. If by the tree creation time, there are quite a number of data objects available, the static creation can be used to set up the $KDB^+ - tree$ based on these data objects. Later, when there are data to be inserted or deleted, we can use the insertion and deletion technique to do the operation dynamically.

The $KDB^+ - tree$ structure has several advantages. In particular, the query becomes more efficient. First, since all the children of a node only differ in one dimension (this node's designated dimension), when doing the comparisons between these children's internal keys and the query range, only coordinates in that dimension need to be compared. But in KDB-tree, coordinates in all the composite dimensions have to be compared. Secondly and most importantly, besides the upper bound, $KDB^+ - tree$ always has the lower bound d for the number of children a node (except the root node) may have. On the contrary, the KDB-tree does not guarantee this lower bound. In other words, d may be very small or even zero. This difference results in a big difference for storage and query efficiency between a $KDB^+ - tree$ and a KDB-tree (Suppose both their volumes are V and each node occupies a page):

1. The height of the tree is $1 + log_{d+1}(V/2)$. Therefore the $KDB^+ - tree$ always has a upper bound for its height but the KDB-tree does not (since d may be 0).

2. The number of pages that a random query has to access is the height of the tree. Therefore a $KDB^+ - tree$ random query only needs to access $1 + log_{d+1}(V/2)$ pages but the KDB-tree random query would have to access many more. A bigger problem is that it does not know how many it would need, no upper bound can be given (even not by Robinson himself).

3. Each $KDB^+ - tree$ node (except the root) is guaranteed to be at least half full. The storage efficiency is more than 50%. Any KDB-tree node could be completely empty. No lower bound for storage efficiency can even be given. For a certain amount of information, many more KDB-tree nodes would have to be accessed. More page accesses mean more I/O operations which mean more time will be spent.

4. For a range query, suppose M data objects will qualify and be retrieved. The approximation of nodes (pages) being accessed is O(M/d). The $KDB^+ - tree$ has such a d but KDB-tree's d could be 0. The $KDB^+ - tree$'s range query is also much more efficient.

The operations applied on the $KDB^+ - tree$ are quite different from those on the KDB-tree. All of them are made to present the $KDB^+ - tree$ a better structure and therefore a better query performance. For the creation of the $KDB^+ - tree$, we introduced the concept of static creation. Other operations such as splitting, concatenation and redistribution, insertion and deletion are all quite different from the notions used for the $KDB - tree$. Their algorithms are based on static creation, which becomes a major factor to differentiate them from the operations used for KDB-tree. As for most databases, the data objects are static, the combination of the static creation and the other algorithms cannot only make the creation of the tree faster and make the tree dynamically maintainable, but can also make the tree more symmetric than Robinson's KDB-tree and make the query more efficient, which is the most important factor for an indexing mechanism. In the next chapter, we will demonstrate on our VPD system that the $KDB^+ - tree$ has a superior query performance.

Chapter 5

Spatiotemporal Query Optimizations

In Chapter 3, we discussed various factors affecting the query speed of an Objectstore application. Here we further examine the index problem for Objectstore, which has not been completely solved. Objectstore is not able to choose the correct strategy for using index in query time. In the last chapter, we introduced $KDB^+ - tree$ as a multi-key index and presented the associated algorithms. In this chapter, we create a $KDB^+ - tree$ as an index for the Objectstore crime database VPDCOL to optimize the VPD system's spatiotemporal query processing. At the end of the chapter, we introduce the third version of the VPD system Query Processor, which uses a combination of $KDB^+ - tree$ index and the Objectstore system index and we argue that this hybrid version has both of their benefits. All experiments in this chapter continue to make use of the query optimization techniques described in Chapter 3.

5.1 Creation of $KDB^+ - trees$

In the $KDB^+ - tree$, pointers to the data objects need to be used. However, like most relational DBMSs, Sybase does not provide users with the addresses or pointers to data records. As an object-oriented database DBMS, Objectstore not only allows this but also provides quite efficient data referencing and de-referencing operations. Therefore it is only possible to implement the $KDB^+ - tree$ on the Objectstore platform.

The Objectstore crime data collection VPDCOL has 148280 data objects of class VPDDATA. Since all the data items are available for use, we use the *static creation* algorithm to create the $KDB^+ - tree$ as an index for the data in collection VPDCOL. As the data set VPDCOL will remain unchanged throughout all tests, no update is needed for the $KDB^+ - tree$ index.

We created two $KDB^+ - trees$ for VPDCOL. One is a 2 dimensional tree over attributes *xcoord* and *ycoord*, the spatial elements of the data. We call this tree $OS2DB^+ - tree$. The other is a 3 dimensional tree over spatial-temporal attributes *xcoord, ycoord* and *comptime.* This tree is called $OS3DB^+ - tree$. We will use these two trees as indexes to execute some spatial-temporal queries and compare the results with those using the system-provided indexes from Sybase and Objectstore.

In Objectstore, users do not have the control to page level, that is, users can not specify the contents of a page. All paging management is manipulated by Objectstore system itself. Therefore, we cannot use the page size to decide the order of the $KDB^+ - trees$. Having done some tests and comparisons on different order values, we made both of our two trees have order of 14. Thus, each of their nodes (except the root nodes) can have 15 - 29 sub-nodes. In the creation procedure, we control this number to be around 22. 22 is in the middle of 15 and 29, and hence the tree is left in a status that it can handle more possible insertions and deletions of data objects with less structural changes (assuming the future insertions and deletions are random). A $KDB^+ - tree$ index for 148280 data items would have 4 levels when each node has about 22 sub-nodes.

The procedure of the query process using the $KDB^+ - trees$ is also composed of 4 steps: *database open*, *query*, *retrieval* and *database close*, like the procedure using the standard Objectstore query functionality. While other 3 steps experience little modification, the *query* step is completely changed. It is done here without using Objectstore query facility and Objectstore index. In the *query* step, we use the $KDB^+ - trees$ to search for all the data objects which qualify for the query constraints.

The pointers to these objects are put in the result collection. This collection is then sorted and used for retrieval.

5.2 Performance Improvements on Spatiotemporal Queries

In this section, we will use the three example queries that have been used before, QUERY2, QUERY3 and QUERY4, to test the spatiotemporal query processing performance using the $OS3DB^{+} - tree$. The selection conditions of these queries are based on the composite keys of the $OS3DB^{+} - tree$: *xcoord*, *ycoord* and *comptime*. QUERY1's selection condition contains a constraint on *compdatetime*, on which a clustered index is built. As explained and demonstrated in Chapter 3, the performance is considerably good when querying over a clustered index. Therefore *compdatetime* was not used as one of the composite keys for our $KDB^{+} - trees$.

Objectstore provides the concept of *segment*. When data elements are created, they can be stored in a single *segment* in the database. If any one data item in the segment is fetched from the database into the client cache, then the whole segment is fetched into the cache. Each of our $KDB^{+} - trees$ is stored into one segment. Whenever the first search into the tree is issued, the whole tree will be brought to local cache so the consecutive query searching will happen locally.

We still use the *warm-up* query WQUERY introduced in Chapter 3 as the first query to "warm-up" the client cache. Its execution will bring all the data in VPDCOL and the whole $OS3DB^{+} - tree$ into client cache so that the replicate database is loaded completely into the cache.

As we found in Chapter 3 (see Table 3.5), the *database open* and *database close* take little time. As a consequence, we will not list the times used for these two operations, rather we combine them into the "Total" processing time. Table 5.1 shows the query processing time for QUERY2, QUERY3 and QUERY4 using the $OS3DB^{+} - tree$ as index.

Comparing the result in Table 5.1 with the Objectstore performance in Table 3.5

Processing Time (sec.)	QUERY2	QUERY3	QUERY4
Query	1.6	4.2	4.4
Retrieving	5.7	12.9	16.8
Total	7.4	17.7	21.5

Table 5.1: The Query Processing Time for Using $OS3DB^+ - tree$ as Index.

and Sybase performance in Table 3.1, we can conclude that the overall speed of spatiotemporal query processing using our $OS3DB^+ - tree$ is faster than both of the other two. Detailed analysis of Table 5.1 vs. Table 3.5 shows the following:

- The *query* time in Table 5.1 is reduced to 1/6 - 1/10 of the *query* time in Table 3.5. This demonstrates that our $KDB_?^+ - tree$ is a very efficient index structure for spatiotemporal point data. The selection of related parameters, such as the order and level number of the tree, and the creation of the $OS3DB^+ - tree$ also seem to be very successful. Here, only the index tree would have to be searched to get the result collection. In Table 3.5, Objectstore uses the *Index-Search* query method. A fairly big intermediate collection of real data objects resulted from B-tree index would have to be searched. Since our $OS3DB^+ - tree$ contains only 3 attributes' values, actually, each internal node only contains one attribute value, the size of the tree in most cases is much smaller than the size of the intermediate result collection. Furthermore, only one part of the index tree will be searched. All these factors contribute to the large improvement to the *query* speed.
- In contrast, all the *Retrieving* time values in Table 5.1 are larger than those in Table 3.5. This is also due to the query procedure for the $OS3DB^+ - tree$. Here, only the index tree is brought into main memory, i.e. without any data objects. In the experiments of Table 3.5, a large number of data objects are fetched into main memory for the *Index-Search* query. Thus, the *Retrieving* in Table 5.1 has to bring more data objects from disk cache to memory than the *Retrieving* in Table 3.5, although everything is already in client cache.

- The comparison of these two tables shows that the reduction in query time overcomes the increase in retrieving time. The overall performance is improved. This is because in the experiments of Table 3.5, many of the data items are accessed twice, once in the *Index-Search*, once in the retrieving. The use of $OS3DB^{+} - tree$ not only improves the query efficiency, but also reduces the number of data objects which need to be accessed. In fact, it guarantees that any data object need be retrieved at most once.

We have illustrated the improved performance for the spatiotemporal sample queries using the $OS3DB^{+} - tree$. Following, we discuss how to take full advantage of this structure to benefit more potential queries for the VPD system. The $OS3DB^{+} - tree$ is still useful when only 2 of its 3 composite keys are present in the query constraints. Also, non-spatiotemporal conditions in conjunction with spatiotemporal queries can be handled.

5.3 Processing 2D Spatial Constraints

In this section, we will analyze the case when only 2 out of 3 composite keys of $OS3DB^{+} - tree$ are present in a query constraint. A common example would be the spatial query, which only queries over *xcoord* and *ycoord*. We will use the $OS3DB^{+} - tree$ to execute some spatial queries. The $OS2DB^{+} - tree$ is a spatial index built on *xcoord* and *ycoord* and therefore favors spatial queries. We will use $OS3DB^{+} - tree$, $OS2DB^{+} - tree$, Sybase system query and Objectstore system query to execute some sample spatial queries and then compare their performances.

The format of our sample queries is:

```
SELECT ea
FROM vpddb
WHERE SQUARE = (X1, Y1, X2, Y2)
```

X1, X2, Y1 and Y2 are all integers within the X and Y domain, respectively. $Y2 > Y1$ and $X2 > X1$. We use 3 sample spatial queries, SPQUERY1, SPQUERY2 and SPQUERY3. The numbers of data records satisfying the selection constraints of SPQUERY1, SPQUERY2 and SPQUERY3 are 2%, 10% and 20% of the total data number in VPDCOL, respectively.

Table 5.2 shows the performance comparisons of these 3 sample queries:

Processing Time (sec.)			SPQUERY1	SPQUERY2	SPQUERY3
Sybase Query			54.1	66.8	63.6
Objectstore Platform	System Provided	Query	70.2	80.5	86.8
		Retrieving	19.0	14.7	21.8
		Total	90.3	95.8	110.4
	$OS2DB^+ - tree$	Query	3.5	6.7	7.7
		Retrieving	25.0	48.9	54.5
		Total	29.0	57.3	63.8
	$OS3DB^+ - tree$	Query	2.9	6.8	11.0
		Retrieving	29.7	45.7	54.2
		Total	33.7	53.8	66.4

Table 5.2: Query Processing Time for Spatial Queries

From Table 5.2, we can make the following observations:

- The performances of spatial queries using $OS3DB^+ - tree$ are very similar to those of using $OS2DB^+ - tree$. Their *Query* times, *Retrieving* times or *Total* times are all similar. Both of their performances are quite competitive compared to the Sybase performance and superior to those using Objectstore provided query facility. This demonstrates that using $OS3DB^+ - tree$ to query over only two attributes *xcoord* and *ycoord* is also efficient.
- The Objectstore system-provided *Query* time is long because it does not have an index to use (The indexes on *xcoord* and *ycoord* were deleted in chapter 3. Too many indexes will force the system to use the very slow *Index-Intersection* query method). Therefore, Objectstore uses *Linear Search* to perform the queries. This is why we say no combination of Objectstore indexes can be always satisfactory.

- SPQUERY1, SPQUERY2 and SPQUERY3 return roughly the same amount of data as QUERY2, QUERY3 and QUERY4 in Table 3.5, respectively. However, the retrieving time that a query in Table 5.2 takes is much longer than the retrieving time of the corresponding query in Table 3.5. This is because that the data retrieved in experiments of Table 3.5 have better reference locality (resulted from the query on *comptime*).

In this section, we demonstrated that even when not all of the composite keys of a $KDB^+ - tree$ are present in a query constraint, the performance of $KDB^+ - tree$ is still quite good.

5.4 Processing Non-spatiotemporal Constraints

In the following, let us study the case when the query constraint contains attributes besides composite keys of the $OS3DB^+ - tree$. One way for us to implement this is:

1. Separate the whole query constraint into two parts: C1, the spatiotemporal query part that involves in some or all of the composite keys and C2, the rest of the constraint. In the VPD system, the complete query is the conjunction of C1 and C2.

2. Use $OS3DB^+ - tree$ to execute C1 to get intermediate collection I1.

3. Use Objectstore query facility to execute C2 on I1 to get the final result collection I2.

4. Sort the pointers in I2 and do the retrieving and projection.

However, this approach would be quite time-consuming because both collection I1 and I2 will be completely searched. Moreover, the data in I2 would be fetched into memory and accessed twice (once in step 3 and the other time in step 4). To reduce this overhead, we can combine Step 3 and 4 into one step, which would be:

3. Sort pointers in collection I1 and search through I1, for each data object which can satisfy C2, retrieve it and perform the required projection.

This new step 3 combines the condition check of C2 and projection together so that one linear search on I1 will be sufficient. As the Objectstore query facility is not able to carry out this kind of combination, we provide our own query facility. We make use of the C2 part of the *parse tree* generated by the *Query Parser*. Each data item in I1 will be checked using C2 and those which return TRUE for the condition C2 will be projected over the required attributes. This approach has been used by many applications, one of the examples can be found in [Wu92]. Since this is not the focus of this thesis, a detailed description will not be given here. This user-defined query facility, called the *secondary checker* here, will be implemented in the VPD system in the future. Here, we will give simulated experiments to show that adding the checking of constraint C2 into retrieving will not affect the query processing performance much.

The sample queries used for the testing originate from QUERY2, QUERY3 and QUERY4. We add the following selection criteria to each of the query as non-spatiotemporal condition: '(compcod = "HITRUN" OR compcod = "THEFT") AND weekday = 5'. The meaning of this condition is "the crime type is either hit-and-run or theft and the crime happens on Friday". The three new queries are named as QUERY2A, QUERY3A and QUERY4A, respectively. Their format can be expressed as:

```
SELECT ea
FROM vpddb
WHERE SQUARE = (X1, Y1, X2, Y2) (compcod = "HITRUN" OR
compcod = "THEFT") AND weekday = 5
WHEN EVERY "T1" - "T2"
```

For QUERY2A, QUERY3A and QUERY4A, the values of X1, X2, Y1, Y2, T1 and T2 are the same as in QUERY2, QUERY3 and QUERY4. The results of the execution of the 3 new queries QUERY2A, QUERY3A and QUERY4A are shown in Table 5.3.

Comparing Table 5.3 with Table 5.1, we see that the times used for the corresponding operations are quite similar to each other. The addition of the non-spatiotemporal

Processing Time (sec.)	QUERY2A	QUERY3A	QUERY4A
Query	3.2	3.7	6.7
Retrieving	5.2	13.4	15.5
Total	8.9	17.6	22.6

Table 5.3: Query Processing Time Using $OS3DB^+ - tree$ After Adding Non-spatiotemporal Condition Checking into Retrieving

condition checking to the retrieving procedure does not affect the retrieving performance or the overall performance. The reason is that each data object in the collection I1 has to be brought into memory only once in either case, since the checking operation is followed directly by retrieving.

Let us compare the numbers of operations in detail. Suppose that there are N data items in I1, among which, M items qualify for condition C2. In the experiments in Section 5.2, retrieving operation is executed N times. In Table 5.3, the C2 will be checked N times and there will be M retrievals. In most cases, M is far less than N and time used for retrieving is equivalent to the time used for condition checking. Therefore we can see that the additional condition checking is not a large overhead.

5.5 Spatiotemporal Query Processor

The three sample queries QUERY2, QUERY3 and QUERY4 only cover relatively simple forms of spatiotemporal queries. We would also like to discuss the processing strategy for some other types of ESQL queries. Besides square, the VPD system can also process spatial queries over different shapes, such as a circle or a buffer along a certain street. We proceed by first using our $KDB^+ - tree$ index to find out all the data items in the bounding rectangle of the area of interest. Further qualification checks, such as whether the data locates in the circle or buffer, are left to the *secondary checker*.

Currently, we have 3 indexes available for VPDCOL: the Objectstore clustered index on *compdatetime*, the $OS3DB^+ - tree$ and the $OS2DB^+ - tree$. As no more

Objectstore system-provided indexes can be created, in order for the VPD system Query Processor to also be able to process non-spatiotemporal queries efficiently, a non-clustered B-tree index[Come79] can be created for each of the most queried attributes in VPDCOL. The statistics on the distribution of data objects can be kept along with each B-tree so the most efficient B-tree can be selected when there are several available. A *query optimizer* is required to pick up the most appropriate index from so many types of indexes. Its decision strategy is described as:

> **if** a query contains *compdatetime* in its temporal condition, **then** Objectstore index on *compdatetime* will be used;
> **else if** the query contains at least 2 out of the 3 composite keys of the $OS3DB^+ - tree$ and the relationship between constraints is conjunction, **then** the $OS3DB^+ - tree$ is selected as index;
> **else** one of the available B-trees is chosen as index.

Actually, whether to select the $OS3DB^+ - tree$ or the B-tree as the index depends on which index can return the smaller size of intermediate collection, I1. However, it is quite hard to predict the result from the $OS3DB^+ - tree$. Since $OS3DB^+ - tree$ deals with the conjunction of several constraints on 2 or 3 attributes, presumably it returns a smaller size I1. This explains why we have its priority higher than B-tree.

The *query optimizer* is also responsible for dividing the whole query condition into two conjunctions: C1 and C2. C1 is sent to an index to get the intermediate collection I1. Both I1 and C2 are then sent to the *secondary checker*.

If we call the retrieving part of the VPD system a *retriever*, we have briefly defined the third version of Query Processor as the data engine for our VPD system. We name it the *Spatiotemporal Query Processor* (STQP). It is a general-purpose query processor consisting of both system-provided and user-defined query facilities. It is built on top of the Objectstore persistent storage. The STQP can process any ESQL queries yet favors the processing of spatiotemporal queries. It is made up of 3 major parts: the *query optimizer*, *secondary checker* and the *retriever*. The detail of the design and implementation of the Spatiotemporal Query Processor is not going to be discussed here. Its performance, which is our major concern, can be guaranteed from the above tests and analysis. Figure 5.4 illustrates the major components and basic mechanism of the STQP.

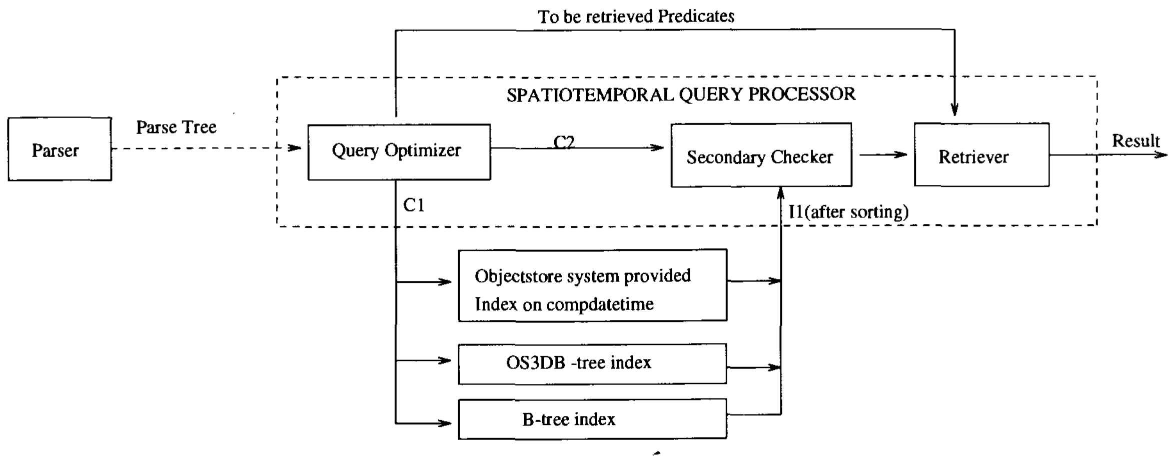


Table 5.4: The Major Components and Basic Mechanism of the Spatiotemporal Query Processor

Compared to its two precedents, the SYQP and the OSQP, the STQP features a higher performance for processing spatiotemporal queries resulted from the use of the $KDB^+ - tree$ indexing mechanism. This was demonstrated by the 50% to 120% performance increase showed by our sample queries because of the use of the $OS3DB^+ - tree$. Therefore, it is the Spatiotemporal Query Processor that can put the $KDB^+ - tree$ into practical use and it is the $KDB^+ - tree$ that brings better performance to the STQP. Objectstore system-provided clustered index and user-created B-tree indexes are used for processing non-spatiotemporal query constraints. This makes the STQP efficient to process any ESQL queries and also solves the index problem brought by Objectstore. The STQP also adopts some behaviors from Sybase, like the selection of B-tree index based on data distribution. It continues to use pointer sorting, which is the VPD system's special feature, to speed up the secondary checking and retrieving. All these features bring better query performance to the VPD system and make the Spatiotemporal Query Processor superior than both SYQP and OSQP, which only use system-provided query facilities.

Chapter 6

Summary and Conclusion

6.1 Summary

The VPD system was designed to process queries about the crime distribution in city of Vancouver for the Vancouver Police Department. Most of the queries to be performed are spatiotemporal queries. The VPD system is composed of 4 major parts: the *GUI*, the *ESQL Query Parser*, the *Query Processor* and its *Databases.* We briefly introduced the GUI and presented a more detailed description for the ESQL query language and the ESQL Query Parser. As a customized query language, ESQL was designed to have more power for expressing spatiotemporal constraints. The Query Parser can parse an ESQL query sent to the VPD system and check its syntax. If it is correct, the Parser retrieves all the useful messages contained in the query and stores them in the *parse tree*, which is then passed to the Query Processor for processing.

The major parts of the thesis concentrate on the performance study of VPD Query Processor. The Query Processor was implemented on two DBMS platforms: Sybase and Objectstore, which represent Relational DBMS and Object-oriented DBMS, respectively. Most of our studies and experiments were made for the query processing on the Objectstore platform, using query performance on Sybase platform as a reference.

In order to compare the performances of these two DBMSs, the same data set and sample queries are used for the experiments on both platforms. The hardware configurations for running the clients and servers of these two DBMSs are also taken into account. After a series of experiments, the characteristics of the two DBMSs are analyzed and several methods were derived to improve the VPD system's query performance on Objectstore.

As Objectstore cannot use index in an appropriate way, we also developed our own multi-key index, the KDB^+-tree for use. Experiments were also done to demonstrate the superiority of this tree index in processing spatiotemporal queries. At last, a new Query Processor, the Spatiotemporal Query Processor, was proposed which combines the advantages of an Objectstore system-provided index and user-defined indexs. A more appropriate query strategy was adopted so that the maximum performance can be achieved for the VPD system.

6.2 Conclusion

One of the major focuses of this thesis is to improve the query speed of the VPD system on Objectstore. As an Object-oriented DBMS, Objectstore has its own specialty that distinguishes it from Relational DBMSs, such as Sybase. Taking advantage of these features makes it possible to achieve a great performance increase for processing the spatiotemporal queries in VPD system. From Table 3.1 to Table 3.4 and Table 5.1, our sample queries' processing enjoys a speed-up of between 35-180 times.

Objectstore allows each object to be accessed by its pointer which is not feasible for Sybase. This feature makes it possible to sort all the resulting pointers before dereferencing them. It also enables us to use object pointers to create KDB^+-tree as a multi-key index to replace Objectstore's inefficient system index. These internal improvements are crucial to the VPD performance.

In terms of the client/server architecture adopted by both DBMSs, Objectstore puts all the query processing on the client site, but Sybase does everything on its server site. This characteristic of Objectstore caused a bad performance at first because for each query all the data needed for it had to be transferred from server to client cache. Knowledge of this special property of Objectstore led us to choose a more appropriate client computer with faster CPU, bigger main memory and client cache for the VPD system and to execute the queries without stopping the VPD system's main process so that the data in the client cache can be reused. In this case, the client cache acts as a replicate database. These reforms greatly succeed in improving query performance as they mimic the nature of Objectstore DBMS.

After all these improvements on both Objectstore's internal query processing mechanism to external configurations, the overall speed of the VPD system on Objectstore becomes much faster than before and also faster than the speed on Sybase, even when the Objectstore server machine is slower and has less memory than the Sybase server computer.

Moreover, our VPD system has proven to be a very efficient query processing system. This demonstrates that designing a customized query language and query processing front-end system for a special application can be a very effective approach. This way, we can generate an efficient and easy-to-use system to fit in users' specific needs, while the user need not be an expert in the OODBMS being used. This will certainly improve the usability of OODBMS in general.

We do not intend that our experiments and comparisons be used to serve as a benchmark. Neither do we wish to compare the quality and system performance between Objectstore and Sybase. We only intend to use these experiments and analysis as a way to discover some of the common factors which can determine the query process performance, and how we can adjust them to speed up the processing. The improvements we made for the VPD system to enhance the performance of general types of queries on Objectstore platform includes:

- Rearranging Objectstore system-provided indexes;
- Sorting pointers in the result collection before retrieving;
- Adjusting Objectstore client machine's configuration;
- Making use of the "replicate database" technique by using client caching.

To improve spatiotemporal query's performance, we relied on a multi-key index tree: the $KDB^+ - tree$, which was for the first time proposed in this thesis. Building a multi-key $KDB^+ - tree$ on the most often queried attributes can reduce the number of pages needed to be accessed and can therefore improve the query performance. As we can see, all the implementations of the changes are very practical. The improvements and principles presented in this thesis can be applied to real life situations and other applications to speed up processing.

Some of the issues proposed in this thesis are not covered in any formal Objectstore documents that we are able to find, such as how the index is chosen and how they are used, when the cache and cache manager are initiated and destroyed. These were discovered by persistent study and thousands of experiments in 15 months, as well as consultation with Object Design Inc. personnel.

Sybase is a very mature and powerful DBMS. Its query facility is proved to be very efficient. Also, Sybase arranges all the processing done in its server site. All the Sybase databases are located at the server site, local to the server. As long as the server computer remains unchanged, the change of client environment does not have much impact on the performance. In this sense, Sybase performance is relatively consistent, as also proved by our tests. The following are some of the issues related to our previous discussion concerning different behaviors of Sybase and Objectstore:

- Both Sybase and Objectstore use client/server architecture. However, Sybase processes all the database calls by its server on the server site. For Objectstore, the real query processing is done on the client site. No approach is strictly better than the other. The Sybase architecture is more suitable for situation in which the DBMS server resides on a very powerful machine with the clients being spread over the network. Objectstore appears to be better when client processes run on powerful computers and the number of consecutive queries is relatively big so that client caching can be in effect.
- Users of Objectstore can use pointers to data objects and de-referencing operations which do not exist in Sybase. This is a definite advantage of Objectstore. However, for queries, Sybase can execute both the selection and projection of a query in one step and return the final result to the users directly. In Objectstore, as we mentioned earlier, a query has to be done in two separate steps. First step is the selection after which a collection of pointers would be returned. Secondly the user has to go through the collection and de-reference all pointers to do the projection part. Usually, this is more time-consuming because all the data in

the result collection would have to be accessed twice, once in the selection period and another in the projection period. Nevertheless, we can take advantage of this property if we can guarantee that the query will be all done by index, without touching the bulky data set. Our experimental results demonstrated this. The multi-key index and the sorting of result pointers, which cannot be provided by Sybase, give a much better performance, as demonstrated by our $KDB^+ - tree$ index testings.

- Sybase uses *Index-Search* query strategy and uses index statistics to decide which index to choose during query run-time. Objectstore makes a bad choice. It adopts the inefficient *Index-Intersection* query method.

Overall speaking, as a young OODBMS which has definite defects, Objectstore is still capable of providing the same or even better associative query performance compared to the successful Relational DBMS, Sybase. The Object-oriented way of data organizing is by no means an obstacle to performance. On the contrary, if we can take advantage of the Object-oriented approach, we can greatly improve the performance, as we demonstrated in this thesis. At the same time, the architecture of the DBMS is also very crucial to performance. This is demonstrated in this thesis as well. In order to achieve optimized query performance, the query strategy has to match the application's requirement and special properties of an DBMS.

6.3 Future Work

Our sample queries can represent most of the queries being used on VPD system now and through the help of these tests, quite a few of techniques are developed for query optimization, which is still fairly adequate for the VPD system at present phase. However, in the Objectstore associative query optimization point of view, we have to admit that the performance testing here is not complete. As we mentioned, our approach is single user and read-only lookup operations on only one collection, concentrating on range queries with respect to spatial and temporal information. Most sample queries only have conjunction as the relationship between constraints. As the development of the VPD system itself, I believe more performance study will be done in the future, such as the performance involving multiple users, multiple collections and join operation between them, "write" operations like insertion, correction and deletions of data, creation and deletion of indexes, a broader range of random and range queries, etc. Through these tests, potential problems are inevitable. More optimization techniques may also be discovered and applied.

Also, our VPD system has a promising future and a lot of further evolution can be done. First, our ESQL can be expanded to cover more functionality and become more powerful and efficient. The GUI shall also be modified to allow corresponding input and display the result of operation. The core part of the VPD system, the Query Processor proposed in Chapter 6 shall be implemented and its function can also be extended to make the Query Processor be able to process more types of query operations more efficiently as the users' requirement goes forward.

Appendix A

GUI Snapshots

In this Appendix, 8 snapshots are given to show the outlook of the VPD front-end system's Graphical User Interface(GUI). The ESQL example query *Example 1* presented in Section 2.3.3 is used to illustrate the inputing procedure of an ESQL query from the GUI and how the final results are displayed by the GUI. Fig. A.1 shows the look of GUI when the VPD system is first started. The GUI is generated as an X window. There are 12 menu buttons on the upper part of the window for accepting user operations. The lower part is the map of city of Vancouver. The map is composed of around 800 enumeration areas (*EAs*). To start the example query, the "Date" button is used to input the starting date of the query, as shown in Fig. A.2. Fig. A.3 shows the inputing of the duration of the query by using menu provided by button "Interval". "Frequency" button is used to input what period of the week is to the query's interest. This is shown in Fig. A.4. Fig. A.5 shows how to use button "Zoomin" to specify the area of the query. After all necessary inputs are done, by clicking on button "ShowMap", the corresponding ESQL query will be formed by the VPD system and displayed in an pop-up window. This is shown in Fig A.6. Clicking the "Input" button in this pop-up window inputs the ESQL query to the VPD system for execution. After the results are generated and sent back to the GUI, the specified area is colored, using different colors to represent different crime numbers occured in each *EA*. This is shown in Fig A.7. Fig A.8 shows the look of the whole map after "ZoomOut" button is clicked. On the right half of the map, there is the color scale showing the number of crimes in an *EA* that each color represents.

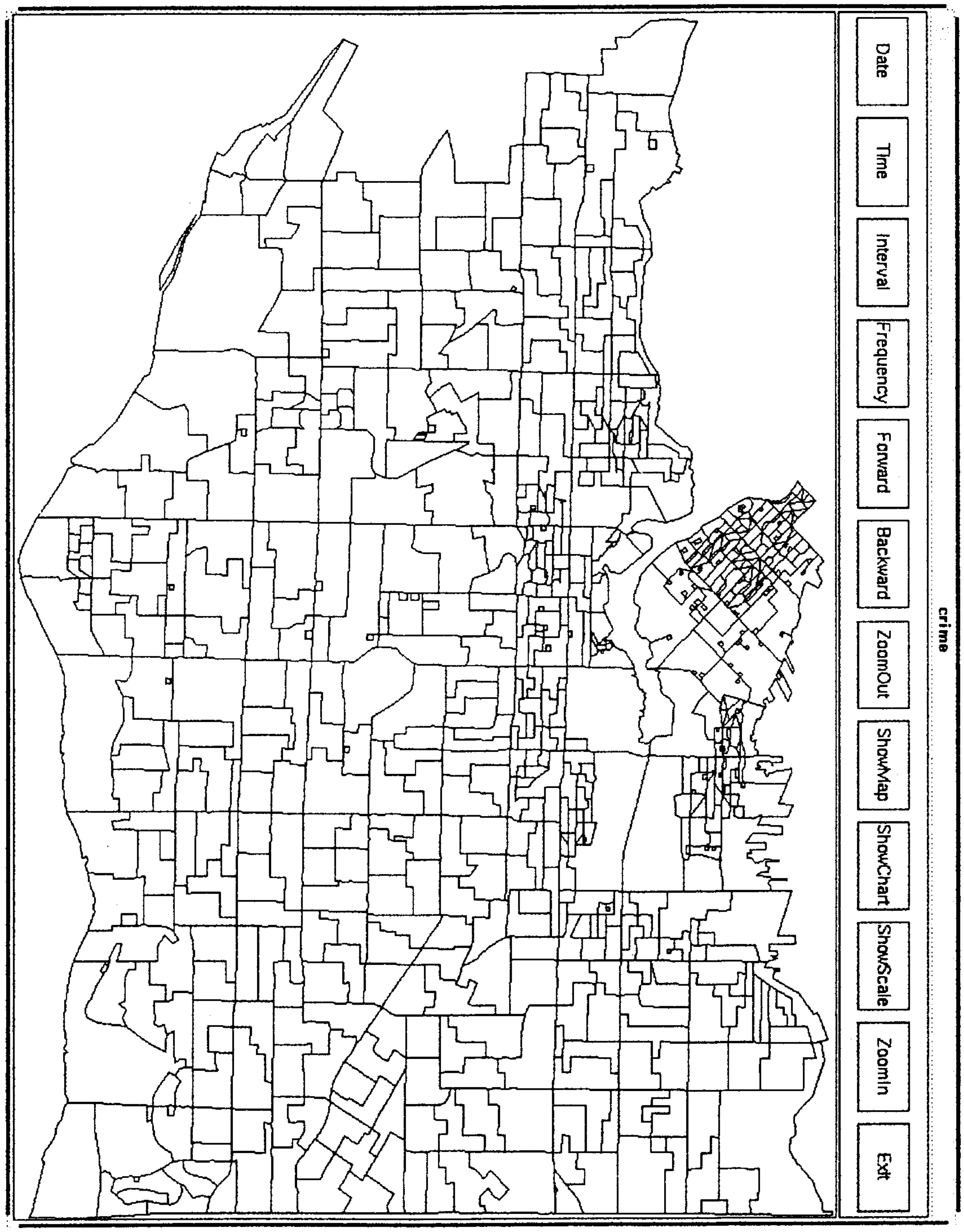


Figure A.1: The Initial Status of the GUI

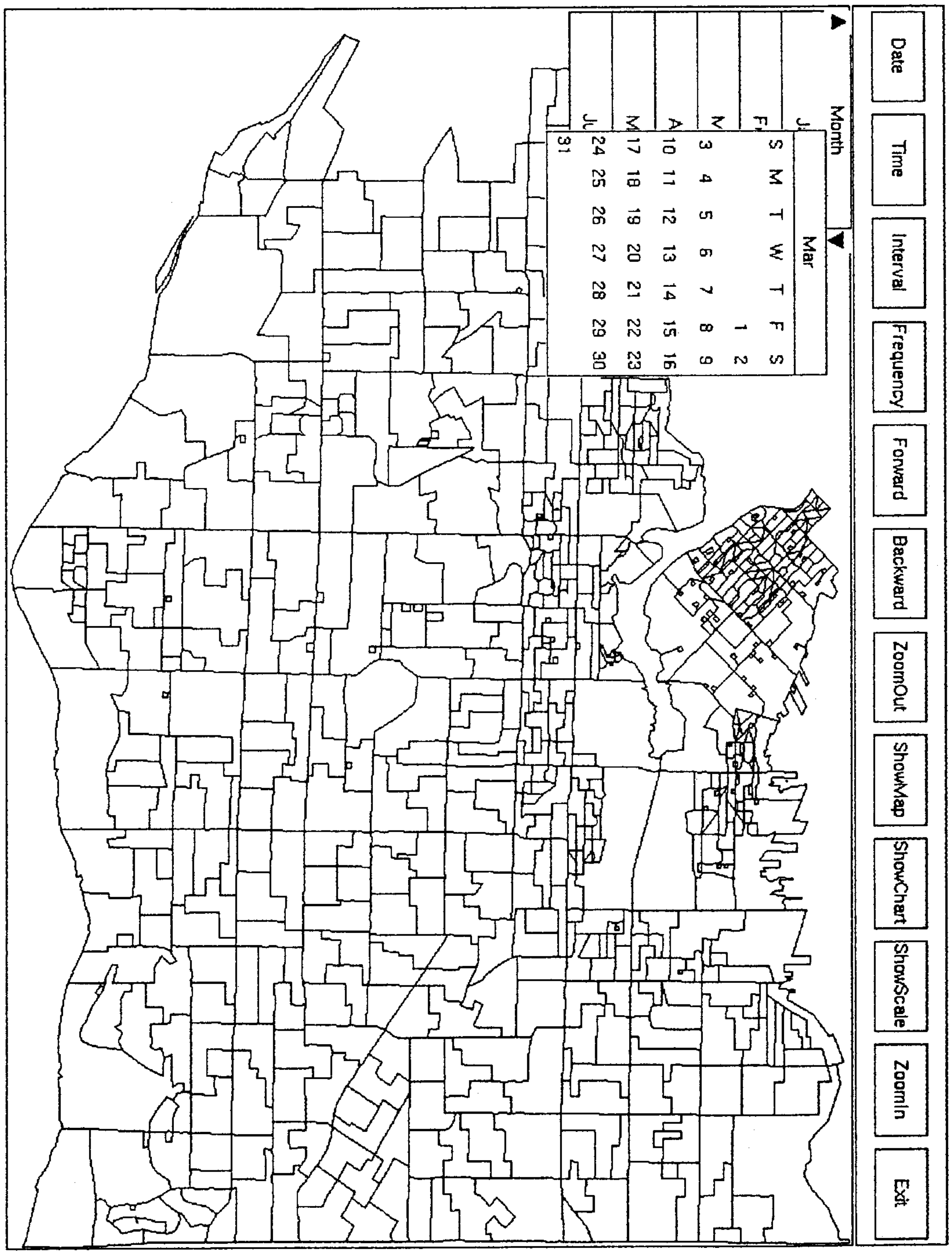
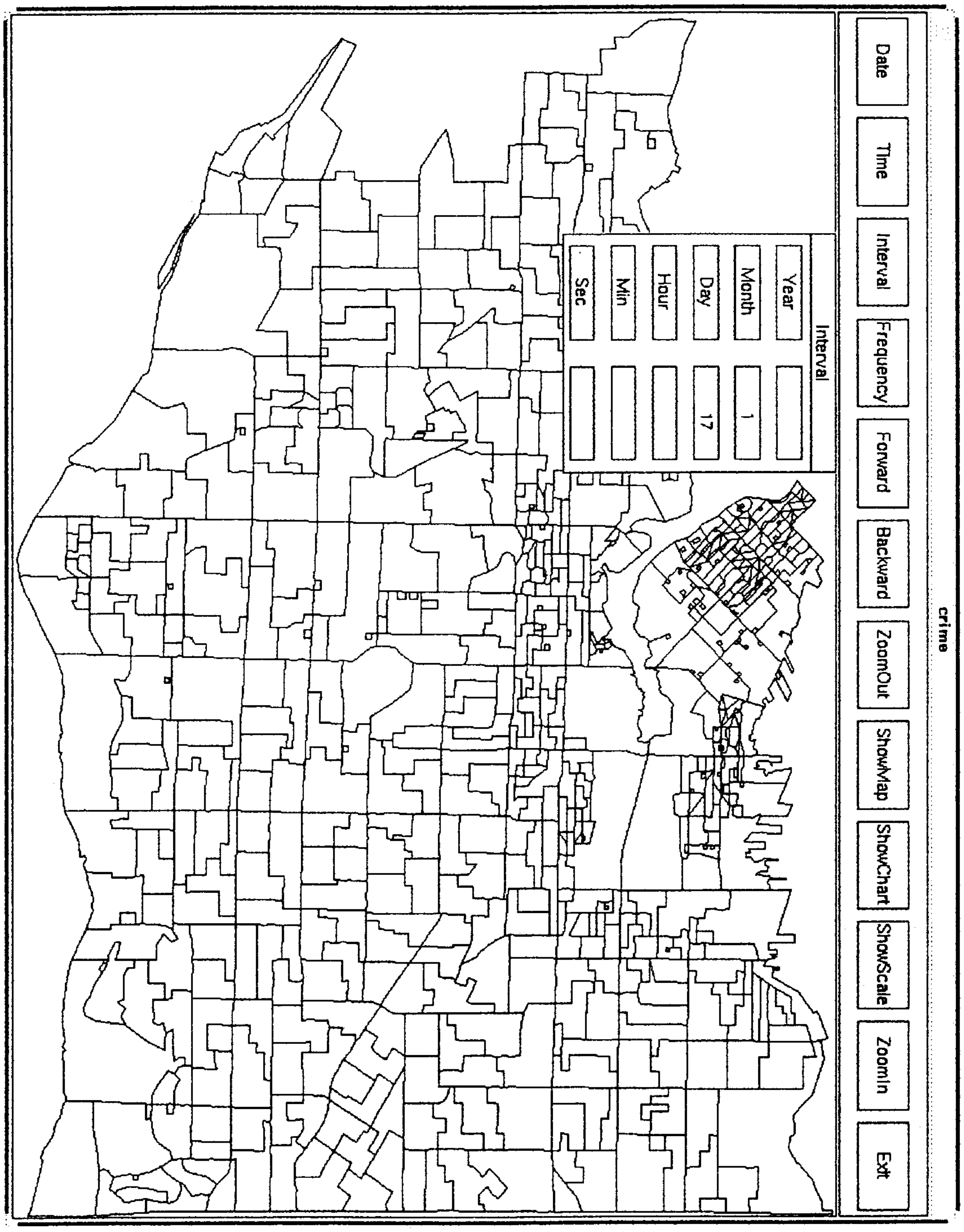


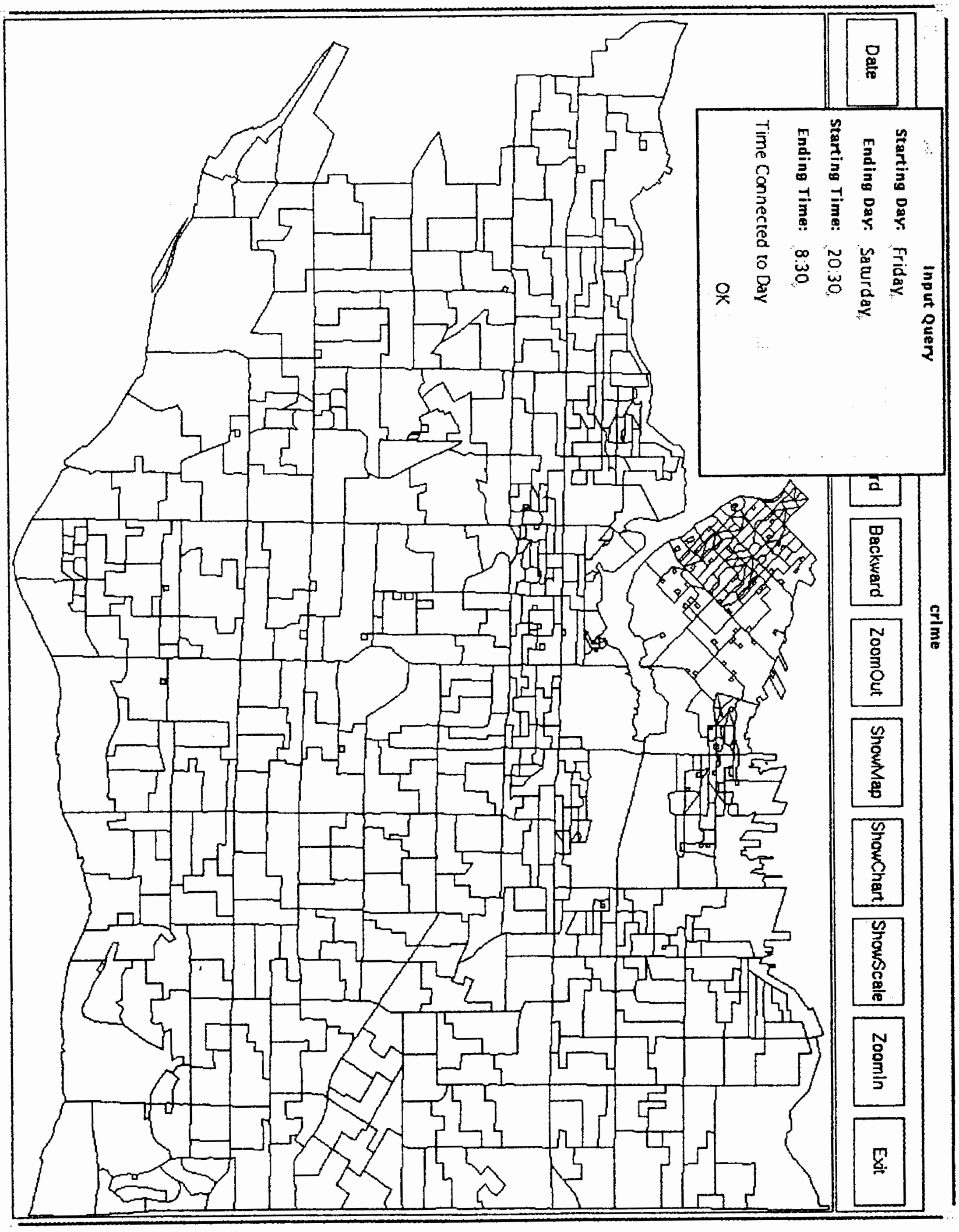
Figure A.2: Inputing the Starting Date of the Query



Interval

Year	
Month	1
Day	17
Hour	
Min	
Sec	

Figure A.3: Inputing the Duration of the Query



Input Query

Starting Day: Friday

Ending Day: Saurday

Starting Time: 20:30

Ending Time: 8:30

Time Connected to Day

OK

Figure A.4: Inputing the Weekday Frequency of the Query

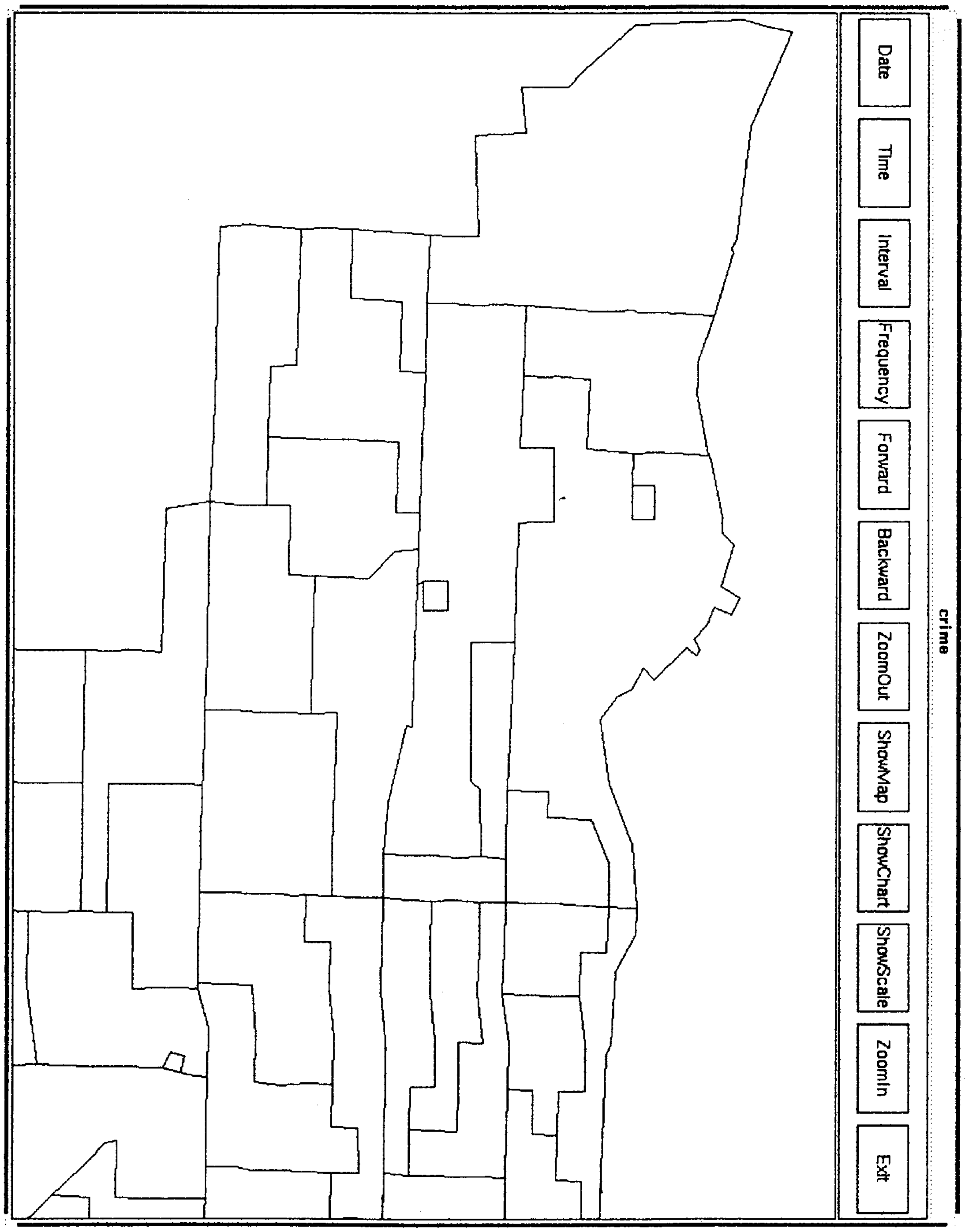
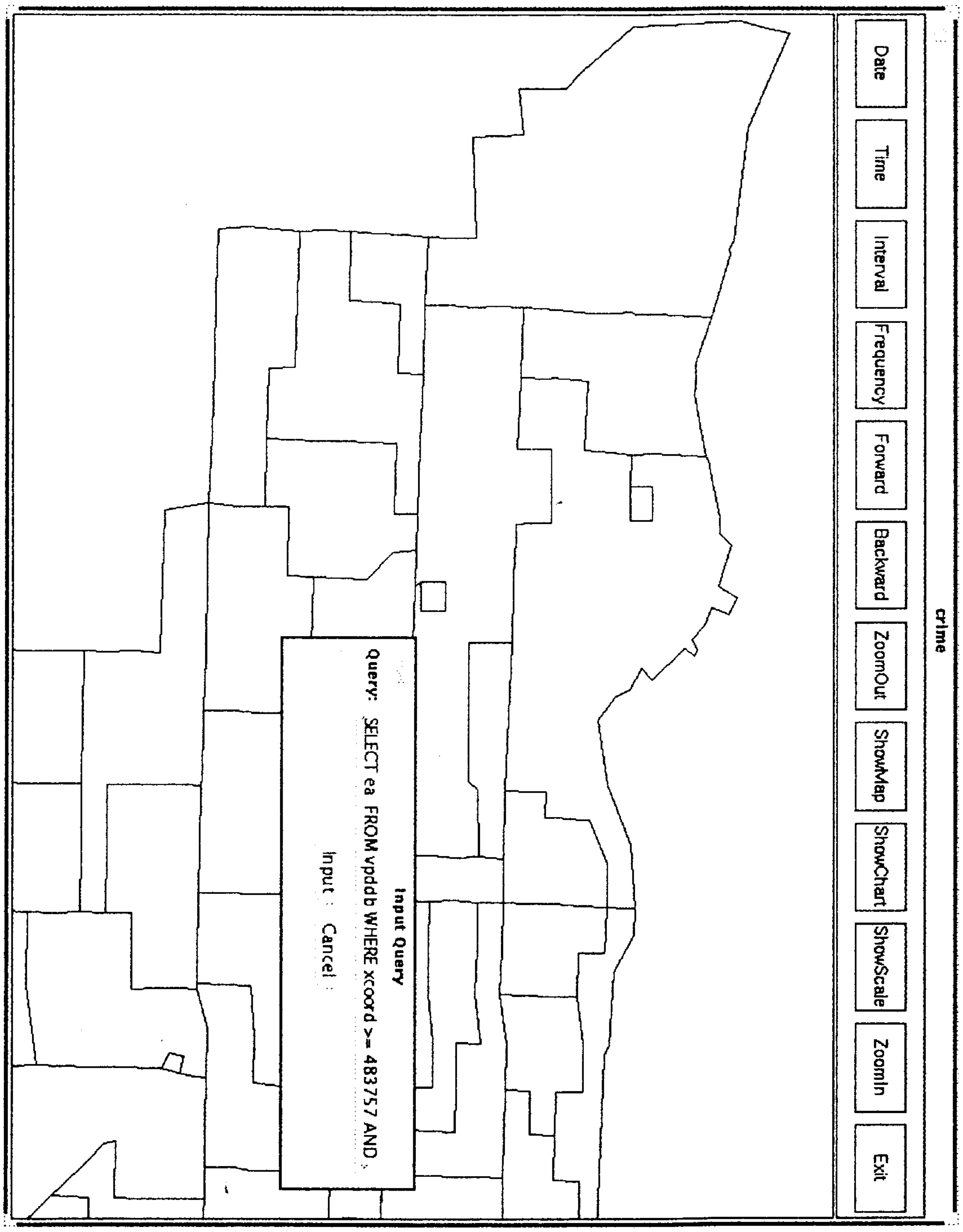


Figure A.5: Specifying the Rectangle Being Queried



Query: SELECT ea FROM vpddb WHERE xcoord >= 483757 AND

Figure A.6: Showing the GUI Formed ESQL Query

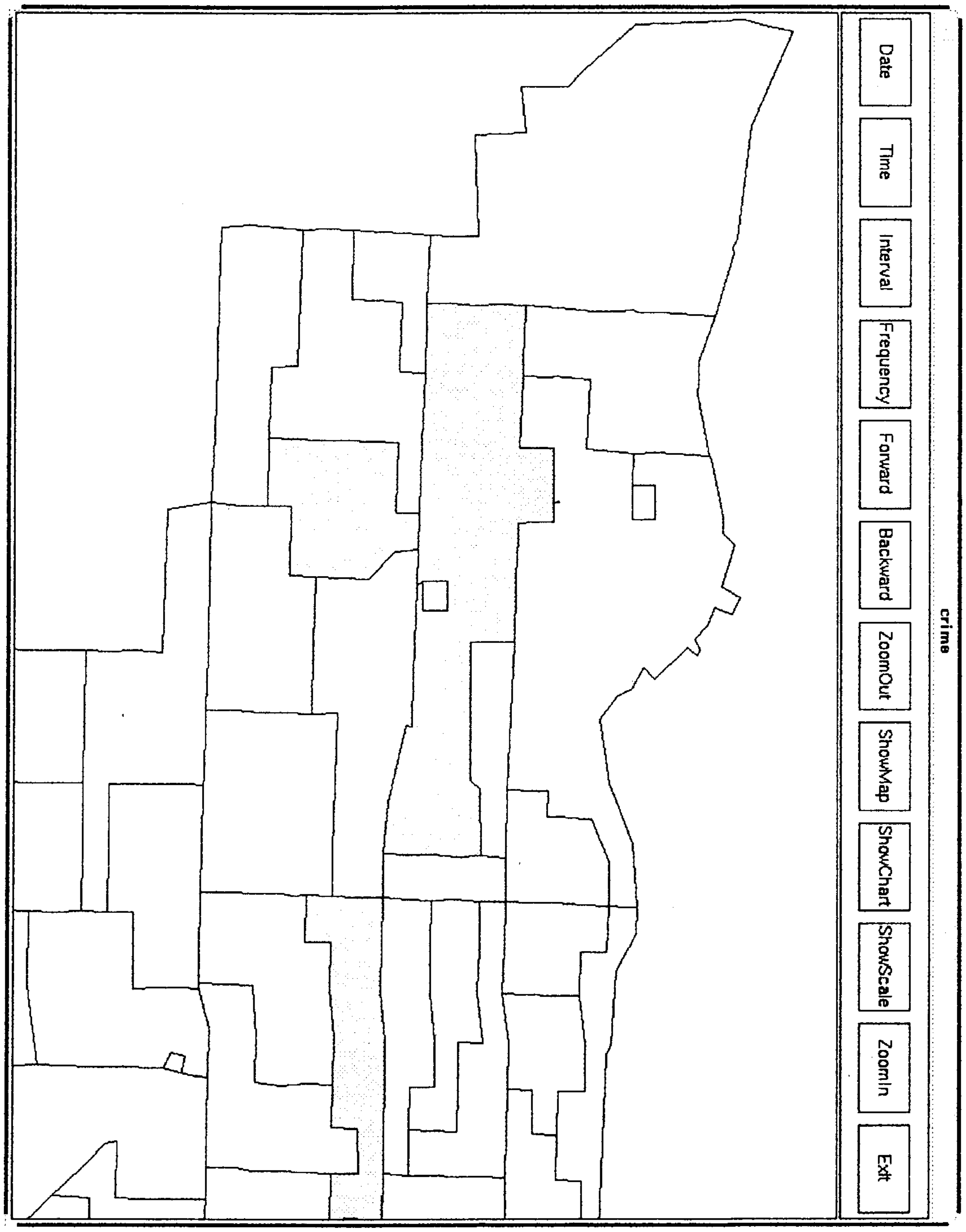


Figure A.7: Coloring the Map According to the Query Result

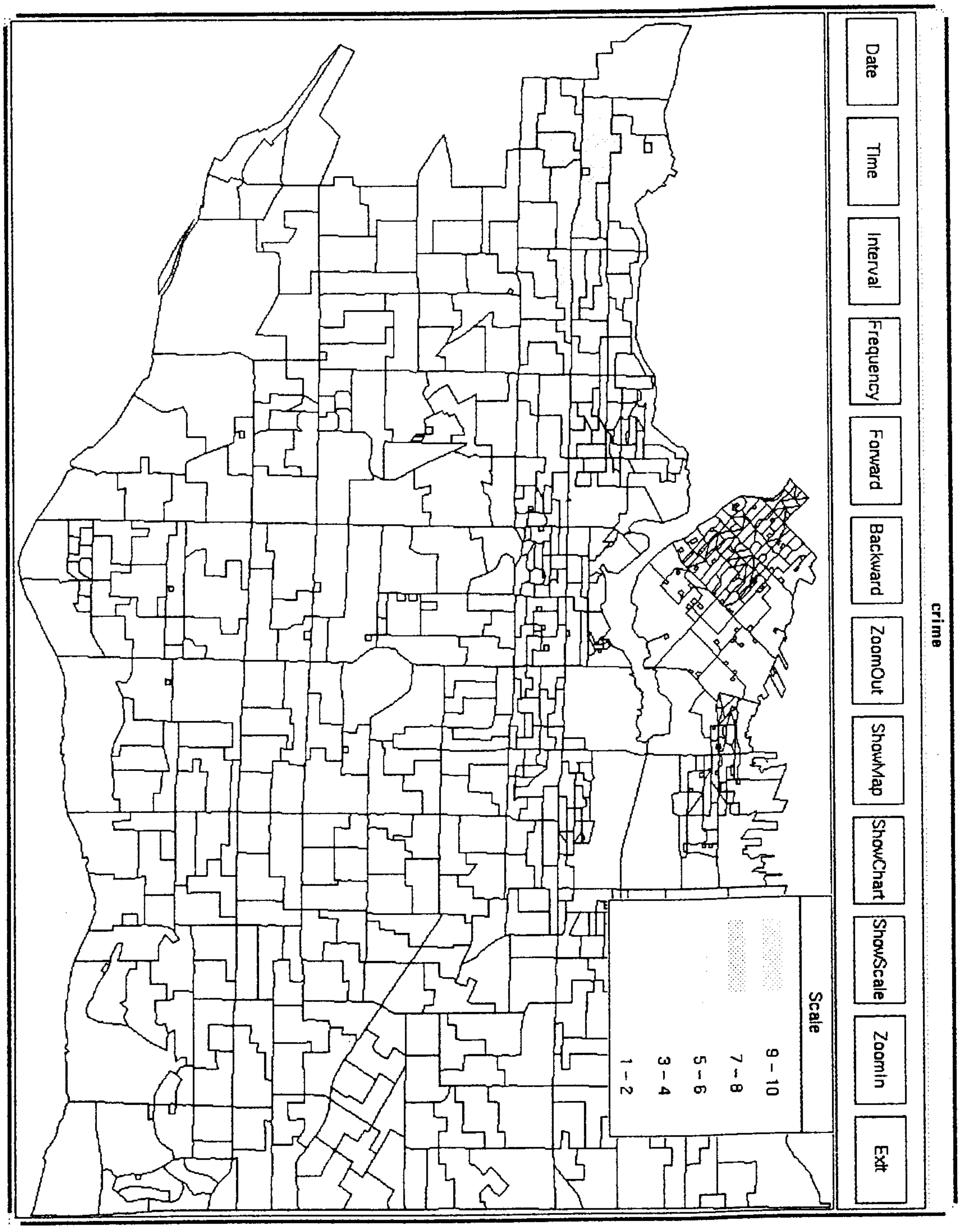


Figure A.8: Showing the Whole Map and Color Scale

Bibliography

[Aref91] W. G. Aref and H. Samet, "Optimization Strategies for Spatial Query Processing" *Proceedings of the 17th International Conference on Very Large Databases*, pages 81-90, September 1991, Barcelona, Spain.

[Bane88] J. Banerjee, W. Kim, K.C. Kim, "Queries in Object-oriented Databases" *proceedings of IEEE Data Engineering*, February, 1988, pages 31-38.

[Bent75] J. Bentley, "Multidimensional Binary Search Trees Used For Associative Searching", *Communications of the ACM*, Vol. 18, No. 9, pages 509-517, September 1975.

[Bent79] J. Bentley and J. Friedman, "Data Structures for Range Searching", *Computing Surveys*, Vol. 11, No. 4, pages 397-409, December 1979.

[Butt91] P. Butterworth, A. Otis and J. Stein, "The Gemstone Object Database Management System", *Communications of the ACM*, Vol. 34, No. 10, pages 64-77, October 1991.

[Catt91] R. Cattell, "An Engineering Database Benchmark", *The Benchmark Handbook, for Database and Transaction Processing System*, pages 247-281, Morgan Kaufmann Publishers, Inc. 1991.

[Come79] D. Comer, "The Ubiquitous B-Tree", *ACM Computing Surveys*, Vol. 11, No. 2, pages 121-137, June 1979.

[Corm90] T. Cormen, C Leiserson and R. Rivest, "Sorting and Order Statistics, Quicksort", *Introduction to Algorithms*, pages 153-171, March 1990.

[Deux91] O. Deux et al, "The O2 System", *Communications of the ACM*, Vol. 34, No. 10, pages 34-48, October 1991.

[Edel86] H. Edelsbrunner and E. Welzl, "Halfplanar range search in linear space and query time", *Information Processing Letters*, Vol. 23, pages 289-293, December 1986.

[Fran83] A. Frank, "Storage Methods for Space Related Data: the Field-tree", *Technical Report Bericht*, Nr. 71, Eidgenoessische Technische Hochschule Zurich, June 1983.

[Fran89] A. Frank and R. Barrera "The Field-tree: A Data Structure for Geographic Information System", *Symposium on the Design and Implementation of Large Spatial Databases*, Santa Barbara, California, pages 29-44, July 1989.

[Gunt88] O. Gunther, "Lecture Notes in Computer Science", Number 337, Springer-Verlag, Berlin, 1988.

[GB88] O. Gunther and J. Bilmes, "The Implementation of the Cell-tree: Design alternatives and Performance Evaluation", *Technical Report TRCS88-23*, University of California, Santa Barbara, October 1988.

[Gunt90] O. Gunther and J. Bilmes, "Tree-based Access Methods for Spatial Databases: Implementation and Performance Evaluation", *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[Gutt84] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *ACM SIGMOD*, Vol. 13, pages 47-57, 1984.

[Kafe92] W. Kafer and H. Schoning, "Realizing a Temporal Complex-Object Data Model" *ACM SIGMOD*, pages 266-275, June, 1992, CA, USA.

[Kim89] W. Kim, J. Garza, N. Ballou, D. Woelk, "Architecture of the ORION Next-Generation Database System", *MCC Technical report, ORION Papers* ACT-OODS-315-89, pages 1-42.

[Kim90] W. Kim, "Object-Oriented Databases: Definition and Research Directions", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 3, pages 327-341, September 1990.

[Lamb91] C. Lamb, G. Landis, J. Orenstein and D. Weinreb, "The Objectstore Database System", *Communications of the ACM*, Vol. 34, No. 10, pages 50-63, October 1991.

[Mark93] V. M. Markowitz and A. Shoshani, "Object Queries Over Relational Databases: Language, Implementation, and Applications", *Proceedings of the 9th International Conference on Data Engineering, IEEE*, pages 71-80, April 1993, Vienna, Austria.

[Meng93] W. Meng, C. Yu, W. Kim, G. Wang; T. Pham, S. Dao, "Construction of a Relational Front-end for Object-Oriented Database Systems", *Proceedings of the 9th International Conference on Data Engineering, IEEE*, pages 476-483.

[Niev84] J. Nievergelt, H. Hinterberger and K.C. Sevcik, "The Grid File: An Adaptable Symmetric Multikey File Structure", *ACM Transactions on Database Systems*, Vol. 9, No. 1, pages 38-71, 1984.

[Obje92] Object Design Inc. "Objectstore User Guide", Object Design Inc.

[Oren92] J. Orenstein, S. Haradhvala, B. Margulies, D. Sakahara, "Query Processing in the Objectstore Database System", *ACM SIGMOD*, June, 1992, CA USA, pages 403-412.

[Robi81] J. T. Robinson, "The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes", *ACM SIGMOD*, Vol. 10, pages 10-18, 1981.

[Same84] Hanan Samet, "The Quadtree and Related Hierarchical Data Structures", *Computing Surveys*, Vol. 16, No. 2, pages 187-260, June 1984.

[Same89] Hanan Samet, "The Design and Analysis of Spatial Data Structures", Addison-Wesley, Reading, Massachusetts, 1989.

[Silb91] A. Silberschatz, M. Stonebraker and J. Ullman, "Database Systems: Achievements and Opportunities", *Communications of the ACM*, Vol. 34, No. 10, pages 110-120, October 1991.

[Snod87] R. Snodgrass, "The Temporal Query Language TQuel, *ACM Transactions on Database Systems*, Vol. 12, No. 2, June 1987.

[Ston91] M. Stonebraker and G. Kemnitz, "The Postgres Next Generation Database Management System", *Communications of the ACM*, Vol. 34, No. 10, pages 78-92, October 1991.

[Syba91] Sybase Inc. "Sybase, Performance & Tuning", Sybase Inc. May 1991.

[Wu92] J. Wu, "Implementation and Evaluation of Dynamic Spatial Query Language", M.Sc thesis, School of Computing Science, Simon Fraser University, September 1992.

[Xu90] X. Xu, "Extending Relational Databases For Spatiotemporal Information", M.Sc thesis, School of Computing Science, Simon Fraser University, July 1990.

[Zhou93] W. Zhou, "How Spatial Data Models and DBMS Platforms Affect the Performance of Spatial Join", M.Sc thesis, School of Computing Science, Simon Fraser University, August, 1993