Implementation and Evaluation of Dynamic Spatial Query Language

by

Ju Wu

B.Sc. Shanghai Jiao Tong University, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

September 1992

APPROVAL

Name:	**Ju Wu**
Degree:	**Master of Applied Science**
Title of Thesis:	**Implementation and Evaluation of Dynamic Spatial Query Language**
Examining Committee:	**Dr. Tiko Kameda , Chairman**
	[illegible]
	Dr. Wo-Shun Luk, Senior Supervisor
	[illegible]
	Dr. Jiawei Han, Supervisor
	[illegible]
	Dr. Ze-Nian Li, External Examiner
Date Approved:	September 10, 1992

Title of Thesis/Project/Extended Essay

Implementation and Evaluation of Dynamic Apatial Query Language.

Author: ____________________

(signature)

Ju Wu

(name)

September 14, 1992

(date)

ABSTRACT

There have been many problems concerning generic languages for object-oriented database systems. However, unlike relational systems, in which SQL has become a standard high level query language with an user-friendly, English-like interface, current Object-Oriented Database (OODB) systems cannot provide a generic query facility. Therefore, we propose to construct an application specific query language (ASQL), which is incorporated with semantics of specific applications or features common to several classes of applications.

A dynamic spatial query language (DSQL) has been implemented in this thesis, which is based on a bi-level data model and OODB system. This implementation shows that how to develop a query processor and optimizer for an ASQL application with impressive performance. Ad hoc queries are processed using compilation strategy and customized query optimization techniques. The DSQL system has an X-Window interface. Version management is implemented to provide a better multi-user environment.

The advantages of OODB systems over the relational systems are also investigated in this thesis. Experiments are performed on two systems, one using ObjectStore (an OODB system) and the other using Sybase (a relational system). Three different approaches are tested on the platform of Sybase, they are the Pure Relational (Normalized), the Less Normalized and the Pre-Loading method. The experimental results demonstrate that the bi-level data model coupled with OODB systems is well suited in ASQL development and the impedance mismatch is the bottleneck for the performance of relational systems. Moreover, the Pre-Loading method provides a reasonable solution to the development of ASQL to overcome the performance problem brought about by impedance mismatch, but there are still problems remaining after adding the Pre-Loading method (e.g. there are no considerations for data recovery and concurrency control).

ACKNOWLEDGMENTS

I am most grateful to my supervisor Dr. Wo-Shun Luk for his thoughtful guidance, invaluable encouragement and great patience in this thesis research. With his help, I have really enjoyed in the thesis work, and have greatly learned from this. Many thanks to Dr. Jiawei Han and Dr. Ze-Nian Li for their great help from the beginning of my master program. I would like to thank Dr. Tiko Kameda for serving as the Chairman of my thesis committee.

I am also very grateful to Kathy Peters and Craig Jones for their kindness of reviewing my thesis.

I would thank my dad, mom and younger sister for their cares and encouragements. Finally, I would thank my dear Hai-Ping for her greatest love.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Application-Specific Query Language

A database query language is a non-procedural language designed for the end user (i.e. casual, non-professional computer user) to manipulate data without resorting to a procedural language. SQL is a standard query language for relational databases, which has a user-friendly, English-like appearance. Nonetheless, most end users prefer to work with programs written by application programmers that run on top of the Relational Database Management Systems (RDBMS), largely because SQL does not operate fully at a level of data abstraction preferred by the end user.

The generic query language for Object-Oriented Database Management Systems (OODBMS) (i.e. not designed for any specific classes of application) is still in the infant stage. In a sense, the Object-Oriented (OO) query languages are extensible, since new operations may be introduced and encapsulated in messages (or functions), assuming here the message is a query primitive. While this scheme might work for simple associative retrieval, it runs into difficulties for complex computations involving objects with different types. Messages must be carefully designed to encapsulate these operations. But there is a more fundamental problem for the current approach of OO query language. Like SQL, OO queries involving set operations are called object algebraic operations. But in typical OO applications, set operations using generic procedures are often inefficient and at times might not even be possible. Take, for example, a Geographical Information System (GIS) as the application domain. Consider the computation task of locating a site which is 5 miles away from object A and 3 miles away from object B. The set-theoretic approach will not work, because it involves intersection of two infinite sets of points (i.e. circles). In addition, the computation must be carried out in consideration of the data representation of points and circles as defined by the application program.

Based on our experiences, it is difficult for OODBMS' to provide a generic query facility that caters to the needs of the end user, while, at the same time being well integrated into the target programming system. For a relational SQL, the same problem still restricts the end users from adding user defined predicates to SQL for complicated queries. We believe that the truly user-friendly query language should be built on top of generic query facility offered by OODBMS', and implemented by what the OODBMS considers as an application program(s). Just as the view mechanism of a RDBMS, it provides an additional level of abstraction. But there are many major differences between a customized query facility and the view mechanism. First of all, the customized query language can incorporate semantic features of a specific application or those features common to several classes of applications, in a way that the generic query language of an OODBMS can not. In fact, this query language should be associated with an application-oriented data model, where data may be manipulated by this language. Thus, there maybe many different query languages customized for different application domains. We call these types of query languages, Application Specific Query Languages. Similar to the Application Layer in the ISO/OSI Reference Model for a computer network architecture, an ASQL provides a level of application-oriented abstraction. Further, the customized software could provide mapping of every possible state of data defined on the application data model into some state of objects defined on the OODBMS', thus, the end user could update the associated data model with the query language. In addition, query optimization is facilitated by the semantic knowledge embedded in the query processor. Despite all these advantages of ASQL, we argue that is not necessarily difficult to implement, and the object-oriented concept embodied in OODBMS greatly eases its development.

The goal of this thesis is to design and implement an ASQL, called Dynamic Spatial Query Language, to convince people of the importance and feasibility of developing an ASQL.

1.2 Dynamic Spatial Query Language

1.2.1 Related Work

Many researches have done work to develop various types of an ASQL (e.g. spatial SQL and temporal SQL.)

Most ASQL's that have been developed are on top of relational database systems (e.g. TQuel is a temporal ASQL developed on Ingres, by extending the traditional relational query language with some temporal features (or predicates)). [14]

Most of the spatial query languages are using a flat table format without much consideration of the hierarchical features of spatial data. An improvement on this was made in the SAND[1] spatial database system, as it stores the spatial attribute in the table along with the non-spatial ones, by providing a link from the spatial attribute to the spatial data structure. This still does not use all of the hierarchical structures that can occur in the applications that we are considering.

Due to the hierarchical structure of the spatial data model, object-oriented database systems match well with this data model and we may separate the spatial data from non-spatial ones at a different level. The major differences of our DSQL from other ASQL's are: **Bi-Level Data Model** and **Object-Oriented Database Platform.**

1.2.2 DSQL and Bi-Level Data Model

We present here a customized query language, called DSQL, for manipulating geometric objects in a three dimensional space. Using DSQL, the end user can retrieve, update and delete spatial and non-spatial information about objects without any knowledge of the exact spatial location of the objects. Our application data model domain is a simplified database, called Block World, which consist of a collection of blocks (e.g. spheres and cylinders) and cuboids in a 3–D space (e.g. room). Various types of operations could be applied to the blocks in Block World (e.g. move and rotate). DSQL can be considered as a high-level interface, whereby, the user can initiate the movement of the blocks and/or interrogate the spatial and non-spatial characteristics of the blocks[9].

DSQL is an English-like, application-specific query language, associated with spatial and non-spatial predicates. The spatial predicates (e.g. on-top-of) define a topological relationship of blocks and the non-spatial predicates (e.g. color) define the characteristics of the block itself.

The data model of DSQL is a bi-level model, block object data model and geometric object data model, to make a distinction between user defined objects and the primitive objects of GIS. Using OODBMS' with some abstraction through its class hierarchy construct, we are able to provide a more elaborate mapping between objects on the two levels. Moreover, the end user may also prefer to leave out low-level, non-essential information about the block objects. Finally, a clearly defined data model at the OODBMS level (i.e. geometric level) will facilitate the task of maintaining, modifying and/or extending the DSQL.

Once we build up the generic GIS query language layer, GIS application programmers will be relieved of the heavy load of manipulating geographical objects. This would allow the query language to be used either interactively or embedded into host language to set up a more specific user interface, providing a similar functionality as the relational SQL. We are trying to provide a development interface for the system programer (i.e. the user often provides the ASQL syntax and application DB schema, we will automatically generate the parsing and evaluating program to do the query processing).

1.2.3 Relational vs. Object-Oriented

Since the DSQL system is a typical engineering application, OODBMS' should be better suited than relational ones, in theory, but we really want to make a comparison between object-oriented and relational systems on this specific application to show OODB systems are better suited.

Object-Oriented Database System Most object-oriented database systems are an extension of existing programming languages, such as C++, to provide persistent, concurrency control and other database facilities [2]. They are often called database programming languages. The common features of these languages are: applications are written in an extended form of an existing programming language, and the language and its implementation (compiler, preprocessor and execution environment) have been extended to incorporate database functionality. These systems typically provide a query language, but both the query language and the programming language execute in the application program environment, sharing the type system and the data workspace [2]. This architecture provides a single unique environment for persistent database objects to appear semi-transparently as transient programming objects.

Object-Oriented vs. Relational Database management systems (DBMSs) based on the relational data model have dominated the market through the 1980s, which is mostly because of its simplicity and strong mathematical foundation of the relational model. However, some of today's data modeling requirements may fail to fit well into the relational framework [3]. Examples are the databases for Computer Aided Design (CAD), Geographical Information System (GIS), Office Information System (OIS) and Artificial Intelligence (AI), in which the representation of complex objects, as well as sophisticated operations is required. Using a relational DBMS in such an environment usually leads to awkward data decomposition and hence to poor performance (shown in Chapter 7). The object-oriented data model allows a natural and direct mapping between conceptual objects and the underlying data model resulting in better performance.

Relational database systems are famous for their simplicity, which makes them attractive because they are easy for end users to understand and utilize. In addition, the model results in mathematically simple structures, and this makes possible a concise and powerful query language that can perform most of the data retrieval and manipulation that typical business applications require. But just because of its simplicity, the tables and the query language, it is far beyond the needs to provide even some basic functionality on complex data models, and thus requires an additional layer to build on top. In this case, binding from SQL variables to host language variables will be established explicitly, resulting in an extra cost. The problems with using two language environments have been widely cited in the database literatures as motivation for database programming languages. The problems have collectively been called the **impedance mismatch** between the application programming language and database query language [10].

Object-objected database systems are best suited in engineer applications (complex data model), compared to any other systems. As Cheng and Hurson point out in their paper, **object identifiers** (an explicit representation of semantic links among objects) and

complex objects (aggregation of heterogeneous objects) are two important features of the object-oriented database system [3]. For example in DSQL, a cuboid is a block object which consists of different kinds of sub-objects, such as center, normal and top face. This allows navigation through semantic links and retrieval of complex objects. Naturally, the most important task for an object-oriented system is to traverse the corresponding graph structure efficiently. Object clustering based on their semantic links becomes a powerful tool to improve the performance. So it is quite natural that object-oriented database systems could provide much better performance in the hierarchical structure than in the relational one, but the object-oriented query language is still in its infant stage. Even still, it is certain that object-oriented system query languages will closely integrate with the host language, hence provide a seamless interface.

Performance Comparisons The promise of object-oriented databases has been that they could potentially provide a much better performance than traditional RDBMS'. As pointed out in Dual and Damon's paper, this is difficult to test because of a lack of benchmarks that are capable of comparing between different types of databases [6]. Many benchmarks, like the Sun benchmark[6], did not involve data clustering, which is the most important factor in performance, and each individual benchmark strictly called for objects to be randomly selected and operated upon. Objects could only be clustered with objects of a similar type, which is seldom true in most engineering applications, and could be kept in the cache over the entire benchmark. In many engineering applications closely related objects are accessed successively, with greater frequency and to a much higher degree than are random, disjoint objects [6]. Because of this general usage pattern, semantically linked objects are often physically clustered together in the database. This semantic clustering normally results in much higher performance, since many of the related objects are read ahead and reside in memory improving the overall access times for related objects.

Performance is a major issue in the acceptance of object-oriented database systems aimed at engineering applications and the most accurate measure of performance would be to run an actual application, representing data in the manner best suited to each potential DBMS. The performance running a DSQL query is obviously a best choice to tell the efficiency of different platforms. Accordingly, we will obtain the performance benchmark on a complicated object-oriented system, which will definitely be more meaningful than those either targeted at relational database usages (Wisconsin benchmark [5]) or based on simple object systems (Sun benchmark [6]).

An object-oriented database system is the best suited for GIS applications, therefore we should build DSQL on top of an OODBMS (ObjectStore) which will then provide maximum functionality and performance. But, since not much effort has been made in the performance comparison between OODBMS' and RDBMS' based on a real application system, we are most interested in the result and how to improve the performance on a relational database system to catch up with an object-oriented one.

DSQL queries with a different degree of complexity, including spatial and non-spatial predicates, will be posed to each approach. And fairness to each approach is another major consideration to convince people of the meaningful comparisons.

Improvement of Relational Approaches There are many reasons why object-oriented systems may have much better performance over relational ones in engineering applications, some of the reasons are composite object and data clustering, object identifier and a closely integrated interface from host language environment to persistent data. We will see from our performance comparison the degree to which we could improve the relational performance by applying these features and what will be the limitations

Composite object and data clustering is hampered by the relational normalization rules. In a relational system, due to the purpose of normalization, a composite object should be decomposed into several tables. But in our application environment, composite objects are handled as a whole, so join operations will be needed to link all the information of one composite object from different tables. Joins obviously hamper performance. Another issue is data clustering. Normally in relational systems, the data are clustered by tables (i.e. to retrieve one composite object, we may have to access several tables, and thus more than one disk I/O). To solve these problems, the **pre-join** may be a better choice. Instead of decomposing the data into several tables, we put all the information into one (or two) large table(s). With this approach, we do not have to do join in retrieving records and therefore less disk I/O is possible.

The impedance mismatch is the biggest problem in relational systems. The relational, persistent, data management language SQL is completely different from the host language, translation will be needed to convert the programming transient object into a persistent data representation. Upon receiving the SQL result, an extra copy of data will be required (i.e. from server buffer to programming variables). Thus two copies of data will be the worse case (i.e. from server disk to server buffer and from server buffer to client variables). While, in object-oriented systems, only one copy is needed, from disk to programming variables, due to no impedance mismatch. We may want to get around this problem by **preloading** the object into the client machine once, and manage it by ourselves in the following access. But to manage the object in the client machine, we will have to borrow the idea from object identifiers to access the object instead of by the relational key. A pointer conversion should be provided.

In this thesis, we not only test the performance of a pure relational approach (normalized), but also apply these two improvements on relational approach to achieve better performance.

1.2.4 Query Processing

The feasibility of ASQL depends heavily on the efficiency of query processing. Two kinds of processing strategies are commonly used, **interpretation** and **compilation**. Normally for on-line query processing (interactive mode), the interpretation strategy will be applied, while for embedded programming, precompilation of embedded queries is widely used.

We choose the compilation strategy for query processing, simply because we will later extend DSQL for embedded usage and would not want to see any inefficiency in query processing. Actually, for an embedded query, compilation strategy could gain more efficiency, because the processing plan has already generated during the precompiling.

The **query preprocessing** should only take a very small part of the whole query processing time, otherwise this application specific query language is totally unacceptable. People would rather write the program to fulfill certain tasks efficiently than waiting for a more flexible but time consuming ASQL to be processed. To get around this problem in our ASQL, we will only generate a very small query processing plan which contains the function calls to the function implementer and then use the dynamic linking and loading method to execute these functions to process the query.

Another crucial part in query processing is **optimization**, a query optimizer, with a carefully defined rule for optimization, is applied to generate an optimal processing plan.

Usually, a generic query optimizer may not be very desirable, since the optimization strategies are also application specific. Our goal is to let the user tell the system some basic optimization rules, and the system could then perform the proper optimization based on these.

Another important issue in query processing is the **display time** as the graphical display becomes more popular in almost all the areas. The graphical display of spatial objects will be investigated, as the total query processing time is no longer the only database access time, display time has been proved to take up more time than database retrieval in some cases. We will also involve investigation of graphical display of object in this thesis.

1.3 Thesis Objectives

This thesis extends the design of the DSQL with implementation in [4] and provides a meaningful performance comparison between the object-oriented and the relational DBMS from a specific application point of view.

The things we want to show in this thesis are:

I. Propose the methods to develop an ASQL query processor and customized query optimizer and show how much benefits we can achieved from an ASQL development.

II. Make a performance comparison between object-oriented platform (ObjectStore) and a relational one (Sybase), and try to find out the suitable platform for an ASQL development.

People may worry about the cost for developing an ASQL. With our implementation efforts in DSQL, we could show people how to develop a query processor and optimizer, and to justify the ASQL as being easy to implement with impressive performance.

The bi-level data modeling (i.e. geographic and geometric objects) provide a concrete base for all detailed design and implementation, and seems to be a standard data model for the generic GIS query. The efficiency and user-friendly features of the DSQL depends heavily on the success of this data model.

In order to obtain a meaningful benchmark, we transfer our low level database accessing part from ObjectStore to Sybase, and obtain the comparison results from two different approaches (Normalized and Less Normalized). Importance of Data Clustering and Memory Mapped architecture has been carefully studied, and related performance tests have been done. Based on our experiences, we could show people how much effort should be made to use relational DBMS for object-oriented usage, and why object-oriented DBMS' are more desirable in engineering applications.

1.4 Thesis Organization

Chapter 2 introduces the bi-level data model for Block World, and the database schemata of block objects and geometric objects. Finally, we present the design of the dynamic spatial query language, based on the bi-level data model.

Since the underlying database is ObjectStore, we will describe in Chapter 3 some of the related features of our DSQL system development and the graphical user interface of DSQL.

Implementation issues (i.e. query processing and optimization) are discussed in Chapter 4. We show how to apply a compilation strategy to query processing and discuss how to generate and evaluate a processing plan. To achieve better performance, we emphasize the optimization strategies to obtain an optimal spatial query design. We also mentioned in this chapter the version control of DSQL.

Object-oriented DBMS' have always claimed to have better performance than relational ones in engineering applications. GIS application benchmarks are given in Chapter 5. Different relational models are presented for comparison, each with the related performance results. We will discuss two important factors in achieving better performance from object-oriented DBMS' — Object Identifier and Data Clustering. To improve object access efficiency through OID, the memory-mapped architecture will also be discussed in details.

Chapter 6 is the conclusion and future work.

Chapter 2
Bi-Level Data Model and DSQL

In this chapter, we describe the design of an application oriented query language, called Dynamic Spatial Query Language (DSQL) [9], used for three dimensional (3–D) spatial databases, which keep spatial and non-spatial information about dynamic objects in 3–D space. Applications in GIS, robotics and mechanical CAD are examples which may benefit from this query facility. We create a simplified database, called **Block World** which consists of a collection of blocks (e.g. cuboids, cylinders and pyramids) in a 3–D space (e.g. a room). In the Block World, objects may be subjected to various types of motion (e.g. move and rotate). The purpose of DSQL is to provide a high-level interface whereby the user can initiate movement of blocks and/or interrogate the spatial and non-spatial characteristics of the blocks in the Block World. The chapter mainly describes the research work in the paper [9].

2.1 Bi-level Data Model

Since the bi-level data model is our foundation for the DSQL design, we will have a close look at this data model before going into the spatial query language design.

Consider the diagram shown in Figure 2.1. At the top is the end user who can pose queries using DSQL. The B-Kernel is a software layer consisting of a transient object manager and a query processor. The transient object manager[1] is in charge of unique identifier generation and management of all block objects. The query processor generates the query processing plans for user queries and does all the essential mapping

[1] In ObjectStore, we could use the facilities of persistent set and unique object pointer to fulfil the task, in Sybase we should build the manager by ourselves.

of functions in a query that involve spatial relationships and/or computations of block objects to the geometric object data model. Finally, at the lowest layer is the DBMS (relational or object-oriented) which includes an additional interface module called the function implementer. The function implementer assigns the optimal procedures for carrying out each function in the query processing plan generated by the query processor of the B-Kernel and executes it. It computes new information and interfaces with the DBMS for the invocation of procedures and the actual retrieval and storage of objects [9].

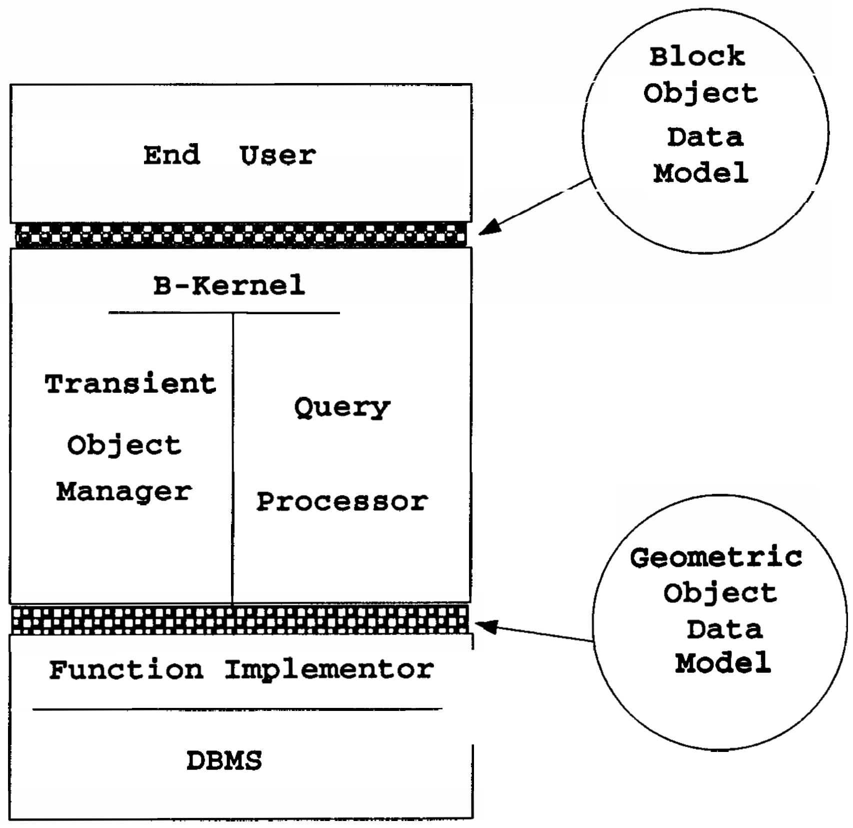


Figure 2.1 **Architecture for Implementation of the DSQL**

The block object data model consists primarily of blocks, a set of semantic spatial functions and some type hierarchies, through which the topological relationship of block objects are defined or derived [9]. DSQL is the language for manipulation of these block objects. The geometric object data model, on the other hand, contains geometric objects which are the actual spatial representation of the blocks and a set of geometric functions that do the retrieval, updating and computing for geometric objects. The DBMS in DSQL could be an object-oriented or relational DBMS. With an OODBMS (i.e. ObjectStore), facilities of its data manipulation could be used directly to construct the geometric functions, while with a RDBMS (i.e. Sybase) an additional layer for converting class hierarchies to relational flat tables is needed.

2.1.1 Block Object Data Model

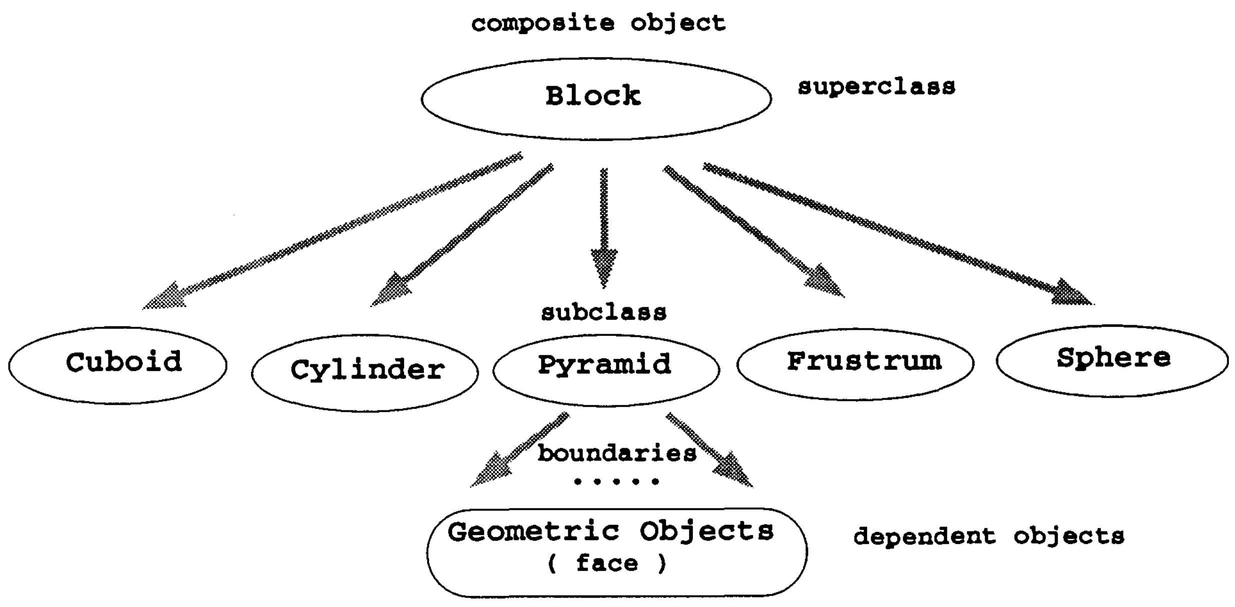


Figure 2.2 **Block Object Data Model Hierarchy**

The Block level data model, which is defined also in boundary representation, is considered to be the end user's interface to the whole system. The geometric level's information (spatial information) is transparent to the end users, except for the colour attribute of faces (non-spatial information). Using the object-oriented model, the spatial functions, such as On_Top_Of, could be more abstract and independent of block type. There are virtual functions in class Block (e.g. Volume and Translate) which means that there will be a run time binding for the actual implementation of these functions according to the particular block type; this is the so-called "polymorphism".

The detail of the data structure Block Object are shown in Appendix A.

2.1.2 Geometric Object Data Model

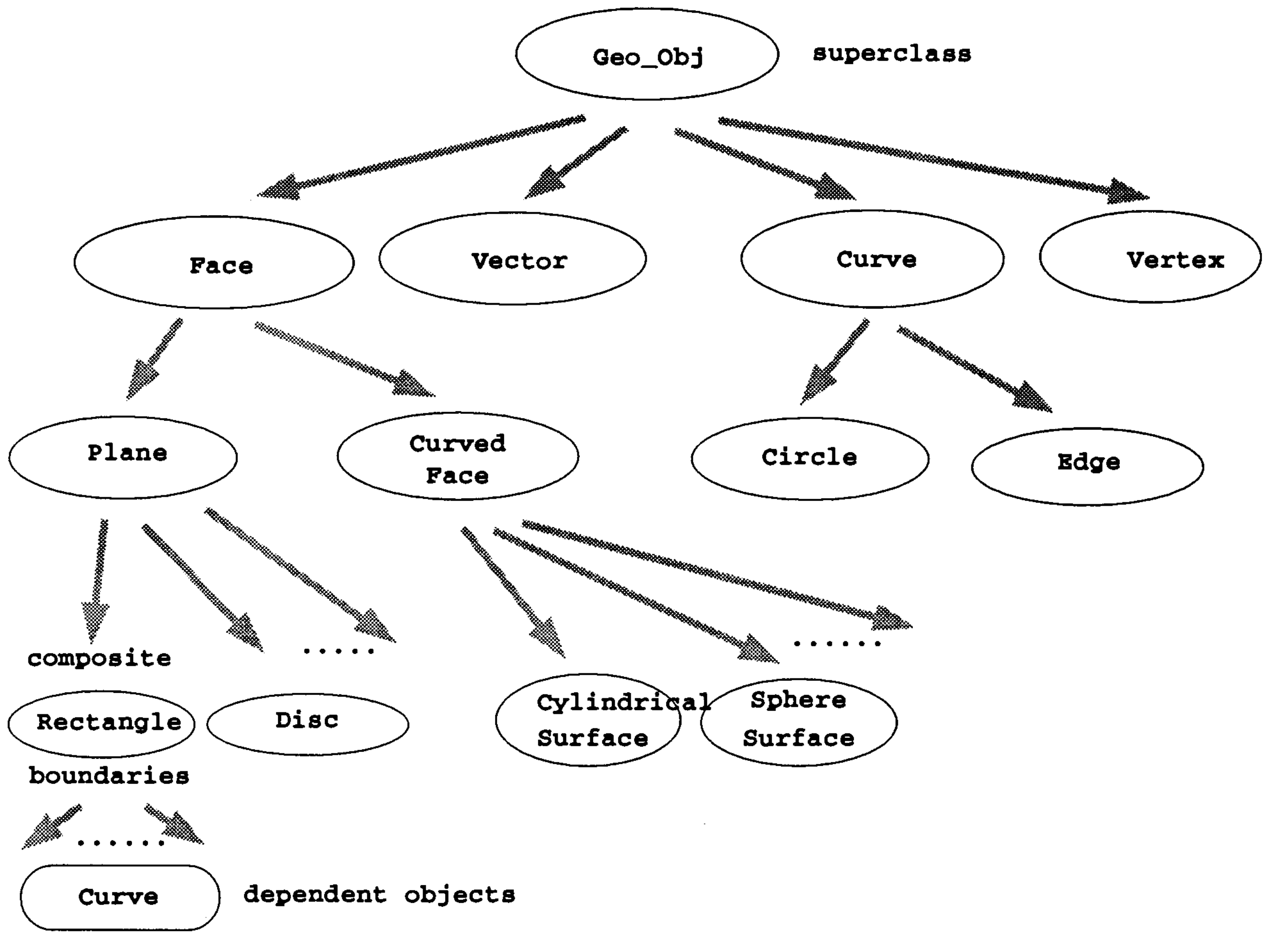


Figure 2.3 **Geometric Object Data Model Hierarchy**

In geometric level object design, one curve is represented by the normal and the length (e.g. edge or circle) not by two end points. The reason why we decided to use this representation (plus the inherited center) is that we are trying to make a level abstraction in each level of boundary (e.g. a cuboid consists of 6 rectangle faces, each rectangle face consists of 4 edges.) If an edge is represented by two end points, to avoid redundancy, there should be another two edges sharing the end points with this one. So when we translate/rotate a rectangle plane, not all of the boundary edges can be translated/rotated accordingly, because there are shared resources (i.e. no distinction between the face and its boundary). While in our schema, to translate/rotate a rectangular plane we only need to translate/rotate the centre and all its boundaries, due to the fact that there is no point where all the boundaries intersect. But this is still a trade-off, we sacrifice the efficiency for elegance (e.g. we need an extra 4 vectors space to store a cuboid).

2.2 Design of the DSQL

2.2.1 DSQL and the Basic Spatial Functions

DSQL is the main user interface of our object-oriented data model used for retrieval, manipulation and computation which could answer spatial and non-spatial queries of a geographic nature. It is used in posing ad hoc queries for different what-if scenarios which are crucial in GIS applications wherein the user needs to specify different conditions that have to be met for a geographic objects to selected. For the design of the Block World query language, we will use a sort of relational SQL syntax, e.g. insert, delete and select, except that we change the update to be several commands with spatial features (e.g. move and rotate). Further we provide several spatial and non-spatial predicates:

I. Spatial Predicates: On_Top_Of, On_Bottom_Of, On_Left_Of, On_Right_Of, On_Front_Of and On_Back_Of.

II. Non-Spatial Predicates: topcolor, bottomcolor, frontcolor, backcolor, leftcolor, rightcolor, type and name.

With the above commands and functions (predicates), an application oriented (Block World) query language will come to life.

In the literature, manipulation of an object is often considered as an operation applied to the objects. We maintain that even though the operation is applied to an individual object, the user should be allowed to select the object for operation, not only by its object ID, but also by some spatial or non-spatial criteria. For example, the user should be able to specify "the block next to that big green sphere" for motion. This representation is of course a database viewpoint. Indeed, a traditional database query language (e.g. SQL) tends to treat the database updating functions in the same manner as retrieval functions. Thus, we include these functions at the query language level, and the details involved in updating the spatial location of object are transparent to the user. The issue is how to design an interface for the user which permits unambiguous mapping into updating functions at the geometric level. We do so with the concept of a virtual object.

Mathematically, there are two kinds of object motions in the space: translation and rotation. Loosely speaking, translation of an object results in movement of the object without modifying the object's orientation. In this thesis, we use a more common term for this translation (i.e. Move). A complete specification of the motion Move requires the knowledge of the spatial coordinate to which the centre of the object in question is moved. This coordinate information can be derived in one of three ways:

I. The value of the coordinate supplied by the user.

II. The position relative to the current position of the object (e.g. move the object 3 metres to the left).

III. A position relative to another object(s) (e.g. the top of a cuboid with a yellow top).

The first option is ruled out since the user is assumed not to have any knowledge of any exact location in the 3D space. To describe precisely the new location (and/or orientation) where the object is to be moved to, we create a virtual object, which is exactly the same as the object, except that it is assumed to be situated in the desired new location. Thus, this new location can be specified via a topological relationship between the virtual object and another object in the Block World, which may even be the object to be moved.

Rotate, as an object motion, is treated in a similar way to Move. Since any rotation of an object in space can be considered to be spatially equivalent to one combination of Move and Rotate about a line through its centre, we need only consider rotations of a block about the x, y and z-axes through the centre of the block. This means that the centre of the block will not be moved during the rotation.

2.2.2 Syntax of DSQL

The following is the syntax of our DSQL query language **Data Manipulation Language** (DML):

```
<Command-Name> <Object-Variable> <Command-Modifier>
WHERE
<Conditions>
GO

<Command-Name> :: = INSERT | DELETE | SELECT | MOVE | ROTATE

<Object-Variable> :: = ID
```

```
<Command-Modifier> :: =
NULL | TO <Object-Variable> | BY NUM METERS |
BY x DEGREES ALONG <coord>

<Conditions> :: = <Predicate-Def> | <Conditions> ',' <Predicate-Def>

<Predicate-Def> :: = ID : <Predicate-List>

<Predicate-List> :: =
<Predicate> | <Predicate-List> LOGIC_OP <Predicate-List> |
'(' <Predicate-List> ')'

<Predicate> :: = <Expression> OP <Expression>

<Expression> :: =
NUM | ID | STRING | FUNCTION_NAME '(' <Function> ')'

<Function> :: = <Object-List> | FUNCTION_NAME '(' <Function> ')'

<Object-List> :: = ID | <Object-List> ',' ID
```

Other features of the query language that are included in a typical query language may be included as well. For example, there can be aggregate functions applied to a set of values, such as Count, Average, Maximum and Minimum. Existential and universal qualifiers are also allowed.

We also allow the version control command in the DSQL. The user can specify from which point we would check out a version of Block World, and create a version history for all the later queries. Further when the user wants to go back to the global version, just stop the version and do a merge; this will be discussed in Chapter 6.

Version Command:

CHECKOUT

MERGE

The following is the **Data Definition Language** (DDL):

```
CREATE ID <Public> <Protected> <Private> <Index> <Supertype> <Subtype>

<Public> :: = PUBLIC: <Attribute> <Method>

<Protected> :: = PROTECTED: <Attribute> <Method>

<Public> :: = PRIVATE: <Attribute> <Method>

<Attribute> :: = ATTRIBUTE ':' <Defn-List>

<Defn-List> :: = <Definition> | <Defn-List> ';' <Definition>

<Definition> :: = TYPE ID

<Method> :: = METHOD ':' <Method-List>

<Method-List> :: = <Method> | <Method> ';' <Method-List>

<Method> :: = <Function> | VIRTUAL <Function>
```

With the help of a DDL, the user could create his own class definitions and store them into a data dictionary. This interface is actually a developer's view of the DSQL.

2.2.3 Examples

Let's consider some examples.

Example 1: Move the cuboid named 'Apollo' to be on top of a cylinder with the yellow top face.

move o1 to o2

where

type(o1)=Cuboid and name(o1)="Apollo" and

On_Top_Of(o3)=o2 and

type(o3)=Cylinder and color(top(o3))="yellow"

go

From Example 1, you may be puzzled at why an additional object is involved. The point is we need a virtual object to define the destination of the move or insert. The function On_Top_Of in Example 1, even though it has the same appearance with those spatial predicates, are actually functions for updating. The semantic meaning of the update function On_Top_Of(o3)=o2 is: the virtual object o2 is on top of o3, but it is to be replaced by source object o1, to put it another way, o1 will be placed to a virtual position (o2) which is on top of o3. The spatial predicate On_Top_Of(o3)=o2 only means o2 is on top of o3.

Example 2: Rotate the cuboid to the left of the block which is on top of a cuboid with a red top face and to the right of a block with a green left face, around the x axis by 90 degrees.

rotate o1 by 90 degrees along x

where

type(o1)=Cuboid and On_Left_Of(o2)=o1 and

On_Top_Of(o3)=o2, and On_Right_Of(o4)=o2 and

color(top(o3))="red" and type(o3)=Cuboid and

color(left(o4))="green"

go

Rotate could be done along any axis.

Example 3: Insert a pyramid to the left of a block which is on top of a cuboid with a red top face and to the right of a block with a green left face.

insert o1

where

type(o1)=pyramid, and length(o1)=10 and width(o1)=8 and

height(o1)=5 and ***bottom_center****=(20.0, 50.8, 0.0) and*

color(bottom(o1))="red" and color(front(o1))="blue" and

color(back(o1))="yellow" and color(left(o1))="pink" and

color(right(o1))="black"

go

Insert could specify a relative position, like On_Top_Of, instead of a bottom_center predicate.

2.2.4 Mapping Block Object Level to Geometric Object Level

There are several types of mapping data modeling constructs between the block object level and the geometric object level. In other words, the following constructs, at the block object level, must be mapped into constructs at the geometric object level, and vice versa: (1) block ID, (2) block, (3) non-spatial function and (4) spatial function and DSQL. For the rest of this section, we will consider each item individually.

(1) Block ID: A block in the Block World is represented by a Vertex object at the geometric level, by the notion of block center. Block center is a sort of block ID, which distinguishes a block from others.

(2) Block: Next, we are going to show how the whole block or its representation at the geometric level, can be recalled given its block ID. Consider as an example a block object which is a cylinder. From Example 1, we know that the function Top maps the block ID into the geometric objects ID of one DISC object. Therefore the top face of the cylinder is easily located. With the boundary representations of these faces, the block can be assembled together from these faces, since each face is stored as an independent object with its own spatial coordinate. Of course, there is an alternative way of representing components of a composite object (i.e. the position of a component being relative to the center of the object (or parent component)). The merit of this scheme is that when the object is moved, and the object is always moved as a whole, only the spatial position of the centre need be updated. In our scheme, the spatial positions of all the components must be recomputed. The main advantage of our scheme is that in computing spatial relationships of the objects, it is very likely that only some components of the objects are involved, and absolute coordinates of these components are essential to the derivation of the spatial relationship (e.g. to place a cylinder on top of a cuboid, one need only to determine a point which is directly above the centre of the top face of the cuboid, at a distance half as high as the height of the cylinder.) For this computation, the knowledge of the absolute location of only the top face , not the whole cuboid, is necessary.

(3) Non-Spatial Predicates: All the non-spatial, block level functions that contain blocks in their input/output function arguments will become geometric level functions, by applying the Center function to all the block types.

(4) Spatial Predicates: Spatial functions at the block level must be translated on an individual basis by the query processor of the B-Kernel into a program that runs on the geometric level. Likewise, a DSQL query will be processed by the B-Kernel to turn it into a program, except that more complicated techniques will be required if efficiency is an important factor. We will only generate a small processing plan which contains function calls to a geometric level function implementer, and use a dynamic linking and loading strategy to execute the calling functions to process the query. This will be discussed in Chapter 4.

Chapter 3
Experimental Environment and Graphical User Interface

In this chapter, we will describe the experimental environment of our performance testing. The database systems used for our experiments are ObjectStore (OODB) and Sybase (Extended Relational), which are two or the most popular commercial systems.

3.1 Systems

All the experiments are done using the client-server architecture. The server is a Sun4/780 CPU server and the client is also a Sun4/780 server with the display set to a local workstation. The reason why we picked a CPU server as our client machine is that it could provide us with larger memory and swap space than some workstations. The workstation used in our experiments is only for graphic display.

On the server side, we have in total 64M of memory and 128M of swap space. The network is a Sun NFS running TCPIP. The client machine also has 64M memory and 128M swap space. The workstation is a Sun Sparcl with a colour monitor.

3.2 ObjectStore

ObjectStore (Version 1.2) is the database system on which we first develop the DSQL system. We will talk about some of its main features.

ObjectStore is an object-oriented database system supporting persistence orthogonal to type, transaction management and associative queries [11]. The data model is not in 1NF (first normal form), as objects may have embedded collections. We will mainly talk about its architecture and associative queries, which are closely related to our DSQL system.

3.2.1 Persistent C++ Programming Language

ObjectStore is designed to provide a unified programmatic interface (persistent C++) to both persistently allocated data and transiently allocated data, with object-access speed for persistent data usually equal to that of an in-memory dereference of a pointer to transient data [8].

With ObjectStore, neither translating nor copying is needed between disk-resident data and programming transient data. Persistent data is just like ordinary heap-allocated data: once a pointer is assigned to it, the user can use it in the ordinary way. This avoids the problem of so-called impedance mismatch. ObjectStore automatically takes care of locking, and also keeps track of what has been modified [8].

ObjectStore makes the interface to persistently allocated objects to support all of the power of the host language (C++). This is in contrast to the traditional data management capabilities of language such as SQL (incompleteness), which are much less powerful than a general-purpose programming language.

The only difference between accessing persistent data and transient data is that persistent data can only accessed within a transaction. While a database is open, transient data can be accessed anywhere within a program.

3.2.2 Memory Mapped I/O

There are a number of very frequent operations in any object-oriented system. Applications send messages to objects, by presenting the system with the logical identifiers of the objects. The system must be very efficient in determining and dispatching the corresponding methods. Furthermore, the system must determine the location of the objects very fast, the objects may or may not reside in memory. This in turn means that, if the objects are on disk, the mapping of logical identifiers of objects to their physical address must be done very fast. We will see how ObjectStore achieves better performance in optimizing the very frequent operation of mapping the OID to physical location.

Tom Atwood separates the architectures of current OODB products into two broad categories :

I. First generation (indirect mapping based).

II. Second generation (memory mapped).

The idea of the first generation architecture is to keep large tables which maintain the location of the objects on the disk. As you need the objects you go through the (hash) tables and get the segment containing the objects in to the main memory. But there are problems:

I. Even if the object is present in the main memory, an access (hashing) takes a minimum of 6–10 instructions to get the object. This is so because every object is indirectly mapped through a table.

II. To keep pointers to the other objects you need a minimum 64 bits rather than the normal 32 bits pointers for the main memory access. The extra bits are needed to indirectly map the objects via the tables.

Being the second generation of OODB, ObjectStore has the memory-mapped architecture, which uses a technique called “pointer swizzling”. In this scheme, all intra-segment memory references are stored as offsets from the segment boundary. All references to objects outside the segment are "swizzled" so that they generate an operating system

(OS) fault whenever they are accessed. Also, each segment keeps track of all other segments which can be referenced by a pointer within this segment. All intra-segment references are done by looking at the offset of the segment in the main memory. However, any inter-segment reference will generate a fault because of "swizzling", and the fault handler will determine the location of the object to be accessed. If the object is in a segment not in main memory, then the segment is brought in the main memory. So, in this approach, all intra-segment references take one machine instruction (rather than 6-10 for the 1st generation.) and on the average (if objects are clustered properly) the performance is very fast.

3.2.3 Set-Oriented Query Language

In relational DBMS', queries are expressed in a special language, usually SQL. SQL has its own variables and expressions, which differ in syntax and semantics from the host language. Bindings between variables in the two languages must be established explicitly. ObjectStore queries are more closely integrated with the host language. A query is simply an expression that operates on one or more collections (e.g. set, list and bag) and produces a collection or a reference to an object.

ObjectStore provides a collection facility in the form of an object class library. Collections are abstract structures which assemble arrays in traditional programming languages, or tables in relational DBMS. Unlike arrays or tables, however, ObjectStore collections provide a variety of behaviours, including ordered collections (lists) and collections with or without duplicates (bags and sets).

This collection enables us to attach a set of object pointers to each DSQL predicate to get the final query result, which is also a set of desired objects.

3.3 Sybase

Sybase is one of the most popular and powerful extended relational database systems. It has a typical client-server architecture, in which the server takes care of most of the work in persistent data accessing.

Sybase has its own buffer management which is independent of the buffer management of the operating system. It has a quite powerful query optimizer for the general purpose query optimization. Buffer management and query optimization are the most important reasons why Sybase can have much faster speed over most of the other relational systems.

The host language can connect to the Sybase server (remotely) using db-library calls, where the client program sends the SQL to the server and waits for the result coming back. Upon receiving the result, the client program will have to bind the information into the host language data representation. Translation of different data representations and an extra copy from the Sybase server buffer to client machine buffer seem to be the bottleneck for complex object processing.

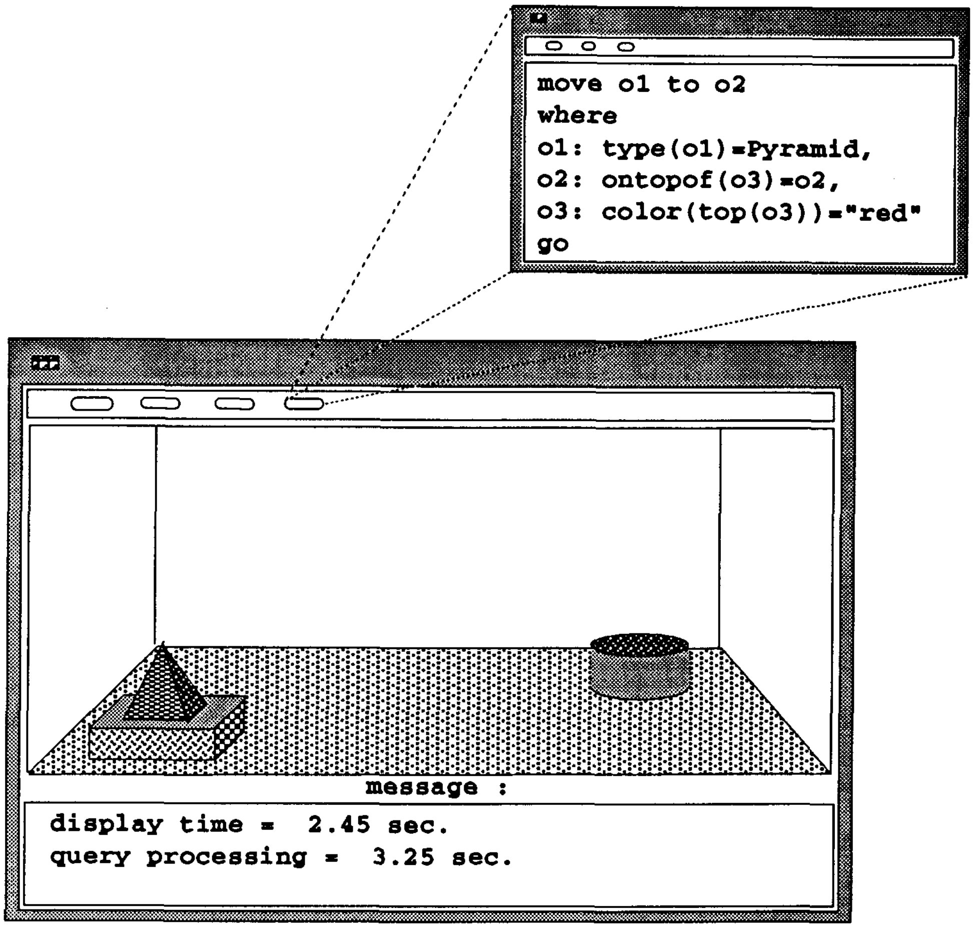
3.4 Graphical User Interface

In this chapter, we will discuss some issues of the graphical user interface in Block World. The Graphical User Interface (GUI) is one the most popular topic in software engineering, and it seems that software could not survive without a GUI. Besides our DSQL based user interface, we build up our GUI using X Windows.

3.4.1 X Window Interface

A very good explanation of the X Window System, taken from Jie Zhang's thesis [19] is as follows: "The X Window System, or X, is a network transparent window system and a widely accepted standard for workstations. Because X permits applications to be device independent, applications need not to be rewritten, recompiled, or even relinked to work with new display hardwares. X is based on a client-server architecture, in which the X server maintains a bitmap display screen on a workstation or an X terminal and distributes inputs from the mouse and keyboard to the X clients. Clients can run locally on the same machine as the server does. However, they can run transparently on remote machines, which might or might not have the same architecture and operating systems as the server's. X is fundamentally defined by a network protocol, the details of which are masked by an interface library called Xlib."

With the above features of X Windows, we can not only create an interface using Xlib and X widgets, but we could also combine with other X Window based graphic packages, from the library level or end user interface level, to build up much more sophisticated graphical user interface. In our Block World, we combined an X window system with a graphic package HOOPS (see HOOP User's Guide for details), X Window based, to build up more a user-friendly interface, shown in Figure 5.1.



```
move o1 to o2
where
o1: type(o1)=Pyramid,
o2: ontopof(o3)=o2,
o3: color(top(o3))="red"
go
```

```
display time =  2.45 sec.
query processing =  3.25 sec.
```

Figure 3.1 **Block World**

In Xlib and X widgets, developing widgets, menu entries, pop-up windows etc. can be generated with great ease, but are lacking in 3–D features. HOOPS, however, has 3–D capabilities. Once we combine these two to build up the interface, we will have nice looking widgets and a 3–D display. Our DSQL graphical user interface consists of a HOOPS display in an X canvas showing Block World scenery, a pop-up window for query input, a message window and some other control menus. Whenever we type in an ad hoc query, all the processing results will emerge through the HOOPS display.

Further, we can also combine our system with some other packages (e.g. SAS) but this connection is not good. Since other systems may not provide a library, they may be accessible only through a specific interface, you could no longer consider it to be part of your system. But with an X Windows architecture, a unique X server will take charge of the display, and you have a window manager standing alone, thus all the X based graphical packages could be combined together to set up a better interface (e.g. you could press a button in Block World window to pop-up a SAS pie chart using an operating system command, even if this package does not reside locally, you could still use remote procedure calls to start it up and handle the network communication as well.)

From the above examples, we could see that X Windows makes it possible to combine heterogenous X-based packages to build up more powerful graphical user interfaces. This seems to be a major trend in the future, because in most of the cases, we would rather use the off-the-shelf package than build up a system from scratch, even though we will have to compromise because of the specificity of the interface of heterogenous packages.

3.4.2 Graphical Display

From the HOOPS manual: "HOOPS is an X-based system for creating interactive graphic applications. Since nearly all graphics applications are database problems, HOOPS is, first and foremost, a database which stores information about which objects to draw, where they should be displayed, and how they should be rendered. The system provides the application program with tools for modifying, querying, searching and displaying the database. The HOOPS database is organized as a tree-shaped hierarchy, resembling a file directory tree. Related elements are grouped together in segments, which are the units of organization within the database. All information stored in the database can be changed; geometry can be edited, attributes can be modified and the hierarchy can be reshaped."

In the DSQL system, we provide a user with a view of the whole Block World, and reflect the changes of the database on the 3–D graphical display, done by HOOPS (e.g. when you type in a query to move a pyramid on top of a cuboid, you will see the corresponding changes emerge on the graphical display.)

In query processing, the graphical display update will be done along with the database update. But there may be problems in the object representation (i.e. the database and graphical package may have a different representation of the object) In order to reflect the database information, data conversion is unavoidable. In our Block World, blocks and geometric objects have a boundary representation, while HOOPS may use a different representation suited for 3–D display. To display a block, we should first extract information of the block object in Block World to construct the parameters used by the display routine in HOOPS, and then have HOOPS decompose the parameters into its internal representation. Let's see the following example for details.

```
void HOOPS_define_cuboid
(id, l, w, h, top_col, botm_col, front_col, back_col, left_col, right_col) {
    create a segment for this id in HOOPS libaray;
    for each face f {
        // each face is defined by 4 endpoints.
        create the child segment for f;
    }
}
```

```
void HOOPS_display(id, center, normal) {
    open the segment of this id in HOOPS libaray;
    rotate it according to given normal;
    translate it to given position (block center);
}
void HOOPS_create_cuboid
(id, l, w, h, top_col, botm_col, front_col, back_col, left_col, right_col, cen, n) {
    // insert definition of cuboid with unique id into HOOPS libaray.
    HOOPS_define_cuboid(l, w, h, top_col, botm_col, front_col,
    back_col, left_col, right_col);
    // linking the cuboid definition in libaray to display at given position.
    HOOPS_display(id, cen, n);
}
void Cuboid::Display() {
    HOOPS_create_cuboid((int) this, Length(), Width(), Height(),
    top->Color(), bottom->Color(), front->Color(), back->Color(),
    left->Color(), right->Color(), block->center, top->Normal());
}
```

There is a translation needed to pass the block information in GIS to the HOOPS display, even though HOOPS also use the boundary representation, but we have to compromise for simple interface in HOOPS display. Another concern is that we will have to recreate a block in HOOPS, whenever we pose an update query to Block World. The reason is is that we should ask HOOPS to reflect the current state of our database, not just show a correct motion on the display. This will create more overhead for the display. When we think of the translation and duplication overhead, we naturally may ask a question as to how we can eliminate this overhead. If we could integrate the HOOPS database with our own database, the overhead will be gone. We will not go into it here, and will only show the weight of graphical display in query processing.

	database retrieval		*3-D display*
	1 non-spatial	*1 spatial*	
select	0.67	19.03	6.24
move	0.58	18.63	9.37
rotate	0.62	19.25	9.81
initialization	30.05		21.64

Table 3.1 **Display vs. Retrieval (On Sun/480 with 64M Memory)**

From the above table, you may find it surprising that display time is already comparable to database retrieval. Thus to improve the performance of the graphical display is another important issue in query processing.

	database retrieval		*3-D display*
	1 non-spatial	*1 spatial*	
select	15.75	35.45	25.76
move	15.28	37.13	25.35
rotate	15.56	37.17	26.04
initialization	53.16		45.23

Table 3.2 **Display vs. Retrieval (On Sun/Sparc 1 with 8M Memory)**

To provide network transparent and device independent features, X Windows has a larger overhead than other local window systems (e.g. SunView) and thus requires a more powerful platform. Since HOOPS is a X-based graphical package, we would like to test how much the performance drops when we run our system on a less powerful platform — Sun/Sparc 1 with 8M memory. You could see from Table 5.2.3, the performance for the HOOPS display drops significantly. Combined with the cost in database retrieval[2], the query processing becomes unbearable. Since X Window is network transparent, we could have a remote X client displaying on the local machine. So, to avoid poor performance, we suggest people use a more powerful CPU server remotely as the client machine for those systems requiring a powerful platform (e.g. X Windows and ObjectStore).

[2] This will be described in detail in the next chapter.

Chapter 4
Implementation of DSQL

In this chapter, we describe the processing of a DSQL query and discuss various strategies to optimize the performance of queries. Some of the strategies are similar to those adopted by relational database systems, such as query compilation and select first [1]. There are other strategies that are specific to the DSQL, or rather to, queries containing functions. Here we consider how these functions are processed when there are OR connectives in the WHERE-clause of the DSQL. We also consider the processing sequence when there are two or more spatial predicates present.

4.1 Outline of Query Processing

Despite the differences in the data models, object-oriented queries may be evaluated in a manner similar to relational queries. This observation is not really surprising. It is well known that the object-oriented database equivalent of the relational selection operation is simply the retrieval of these instances of the target class and the retrieval of these instances requires retrieval of the instances of other classes, which are recursively referenced as values of target class attributes.

In DSQL there are five major steps to process a query: (see Fig 4.1)

I. Pose the ad hoc query to YACC (Yet Another Compiler-Compiler) to generate a parse tree. We use YACC so that we only need to give the BNF of the query language and it generates the parser programming to do syntax and grammar checking automatically.

II. Pass the parse tree to the query processor to generate a processing plan. A query optimizer is involved to obtain an optimal plan for higher performance.

III. Use the Unix system facilities (dynamic linking and loading) to compile the processing plan into a shared library (called query.so).

IV. The file query.so is mapped into a number of function calls to the DBMS. With Unix dynamic linking and loading facilities (dynamic linking editor), plans (functions) in shared library (query.so) could be mapped into system function calls to DBMS.

V. The function calls to the DBMS are processed and the query result is sent back to the end user.

Applying a compilation strategy for the processing of an on-line ad hoc query is often considered a difficult task, but with the help of dynamic linking and loading in Unix, we can process the ad hoc query very efficiently. We have a constant overhead (2 seconds) in query compilation, for most queries.

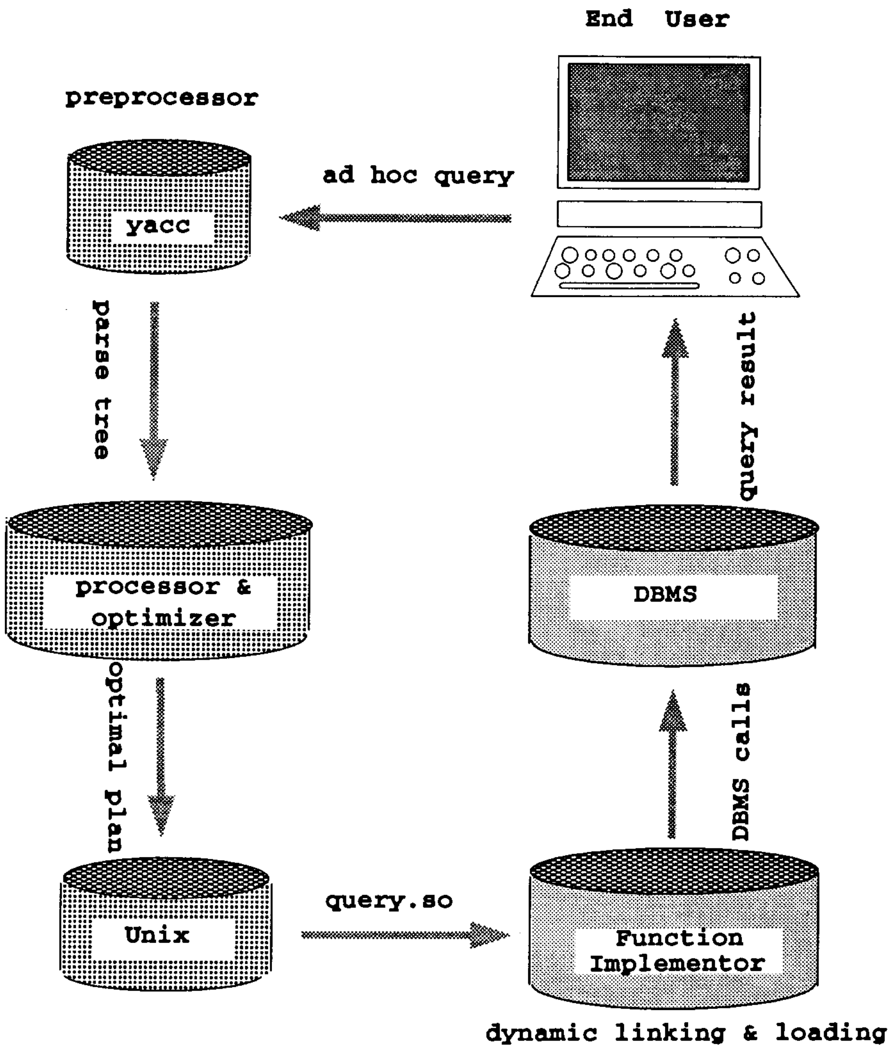


Figure 4.1 **Flow Chart of ad hoc Query Processing**

4.2 Query Processing Plan

The generation of a query processing plan (Step II) has three major steps:

I. Classify predicates by objects: Since each object in the query will have its corresponding predicates, we have to classify all the query predicates by objects.

II. Optimize the parse tree: The position of predicate nodes in the parse tree will be adjusted according to a proper optimization strategy. See section 3 for details.

III. Each of the objects in the query are related, there exists a dependency graph for all the related objects. Query processing order is bottom-up in the dependency graph.

```
GeneratePlan() {
Classify predicates by objects;
For each object do optimization on its associated predicates tree;
Decide the order to process each object;
Generate the query plan;
}
```

Figure 4.2 **Query Processing Plan Algorithm**

Let's give an example of DSQL query and look at the generated processing plan and how to evaluate it.

Example 4.1

```
select o1 where
On_Top_Of(o2)=o1 and (name(o1)="Apollo" or type(o1)=Cuboid) or
On_Bottom_Of(o5)=o1 and On_Left_Of(o3)=o1 and On_Right_Of(o4)=o1 and
color(front(o2))="red" and
color(top(o3))="yellow" and
color(left(o4))="green" and
color(right(o5))="pink"
go
```

This query will select the cuboids named "Apollo" and on top of a red front face block, or below a pink right side block and to the left of a yellow top block and to the right of a green left side block.

First we classify the predicates by object. We may easily see that the first 6 predicates are associated with block o1, and the remaining ones are corresponding to o2, o3, o4 and o5, respectively.

The parse tree for block o1 will be as following, and will be optimized following the rules in the next section.

The processing order will be o2*o3*o4*o5*o1, and the parse tree is shown in Figure 4.3 will be translated to the processing plan. Now let's look at the processing plan, and how to evaluate it efficiently.

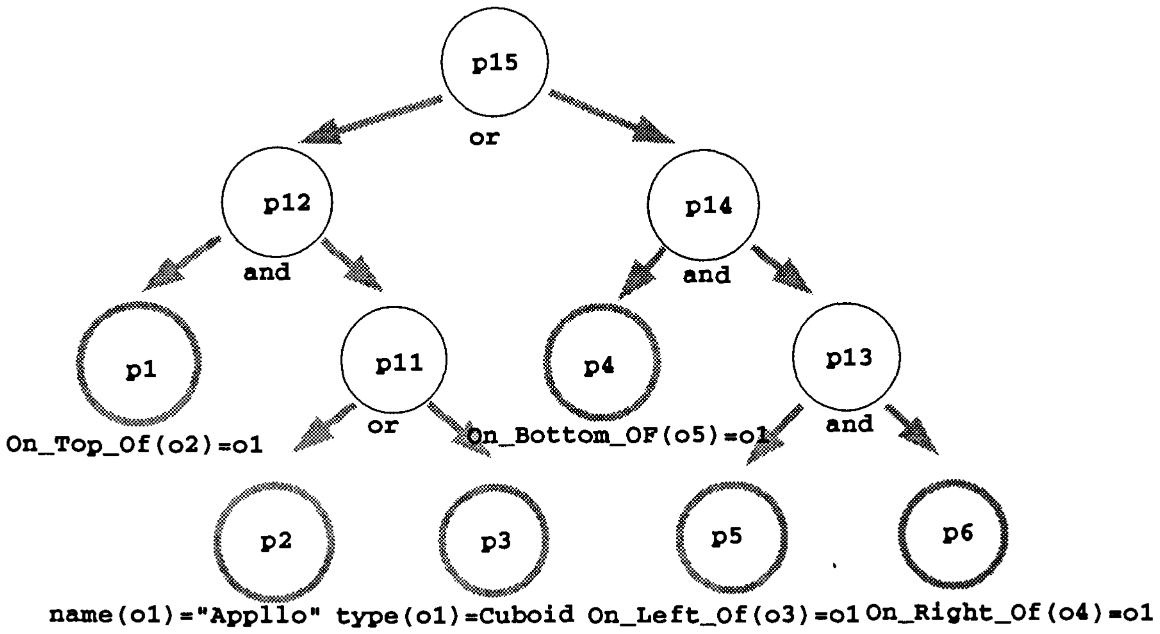


Figure 4.3 **Parse Tree with Proper Symbol Names**

p7: color(front(o2))=”red”;

p8: color(top(o3))=”yellow”;

p9: color(left(o4))=”green”;

p10: color(right(o5))=”pink”.

Processing Plan:

```
void DyCondition() {
      frontcolor(p7, o2, “red”);
      topcolor(p8, o3, “yellow”);
      leftcolor(p9, o4, “green”);
      rightcolor(p10, o5, “pink”);
      name(p2, o1)=”Apollo”;
      type(p3, o1)=Cuboid;
}

void DyQuery() {
      Intersect(p2, p3, p11, o1);
      On_Top_Of(p1, o1, o2);
      Intersect(p1, p11, p12, o1);
      On_Bottom_Of(p4, o1, o5);
      On_Left_Of(p5, o1, o3);
      On_Right_Of(p6, o1, o4);
      Intersect(p5, p6, p13, o1);
      Intersect(p4, p13, p14, o1);
      Union(p12, p14, p15, o1);
}
```

Evaluation:

```
void Process() {
  foreach (a_block, Block::extent)
    mapping to DyCondition() with Dynamic Linking Editor;
  mapping to DyQuery() with Dynamic Linking Editor;
}
```

The final result will be associated with root **p15**.

To evaluate the processing plan efficiently, we try to finish processing all of the non-spatial predicates in one scan (DyCondition();) of the database (i.e. for each block in Block World, test with all the non-spatial predicates, and attach it to those objects with a true value). That is definitely faster than considering these predicates separately. This processing plan has already been optimized, and we will discuss the optimization strategy in the next section.

4.3 Optimization

Before we study the details of query optimization, let's look at the spatial predicate processing function first.

Spatial Predicate: On_Top_Of(o2)=o1 ==> Function void On_Top_Of(o1, o2);

```
void On_Top_Of(predicate p, block_set o1, block_set o2) {
    // retrieve the set from variable name from symbol table.
    foreach (b1, o1)
        foreach (b2, o2)
            if (b1 not on top of b2) then attach b1 to p;
}
```

If we ignore the query optimization, the query can be processed. Let's look at how the following query can be processed.

```
select o1 where
On_Top_Of(o2)=o1 and name(o1)="Apollo" and
color(top(o2))="red"
go
```

Assuming the total number of blocks is N, the number of blocks with a red top face is N/6 (6 faces of a cuboid), and the number of blocks named "Apollo" is 1.

The processing plan will be: process o2, find those blocks on top of o2, and intersect with those blocks named "Apollo".

The processing cost is: N + N*N/6.

However, if we confine all the possible blocks on top of o2 to have name "Apollo", then the processing will drop sharply to 2*N.

4.3.1 Applying Non-spatial Constraints to Spatial Predicates

It is well known that for query optimization of relational DBMS' the most important rule is "select before join". Our spatial predicate (e.g. On_Top_Of(o2)=o1) is a nested loop, and obviously we would like to attach more constraints to o1 and o2 before entering the loop.

Let's look at one example of how to apply constraints (non-spatial predicates) to spatial predicates.

Example 4.2:

```
select o1
where
On_Top_Of(o2)=o1 and color(front(o1))="red" and
(type(o1))=Pyramid or name(o1)="Apollo") and
color(top(o2))="green"
```

go

This query is to find all objects that have a red front, green top face, and is a pyramid or a red front face and is on top of a block with a green top.

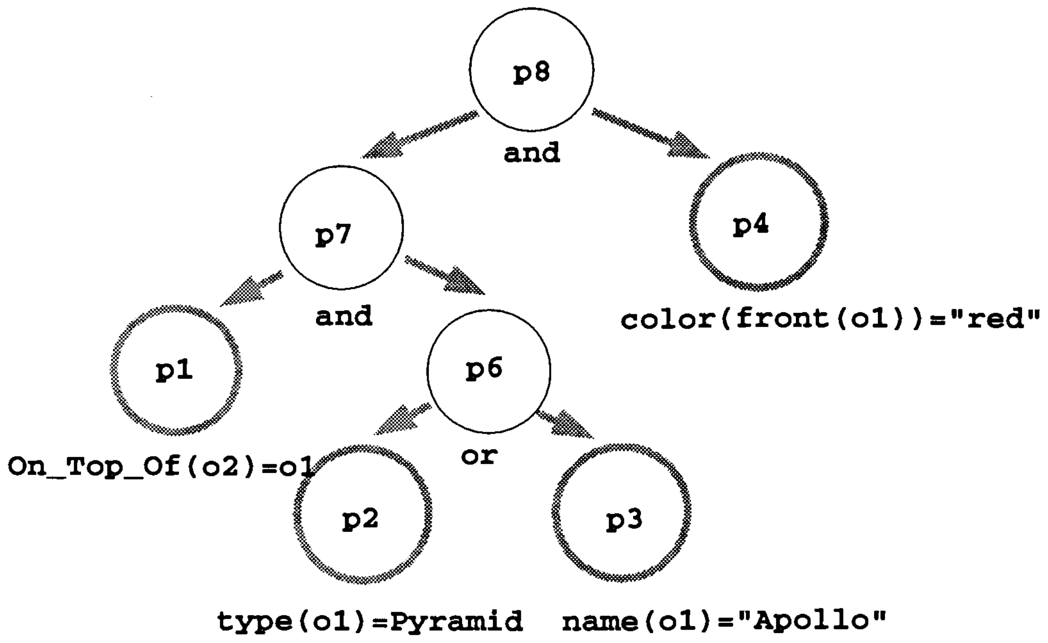


Figure 4.4 **Parse Tree Before Applying Constraints for o1**

First classify the predicates, p1, p2, p3 and p4 to be associated with o1, p5 to o2.

The Logical formula for the predicates of o1 will be like this:

$$p1 \wedge (p2 \vee p3) \wedge p4$$

Without optimization, the processing plan for the above query would be:

I. Process p5 of dependent object o2.

II. Find o1 satisfying p2 or p3.

III. From o1 in II, find the subset satisfying p1.

IV. Intersect o1 with those satisfying p4.

It is obvious we did not apply the constant p4 (color(front(o1))="red") to minimize the size of o1 before testing the spatial predicate (nested loop). We could involve a function called LowerDown to apply the constraints to spatial predicates (i.e. lower down a sub-tree with pure non-spatial predicate(s)). After optimization, the optimal parse tree will be as follows.

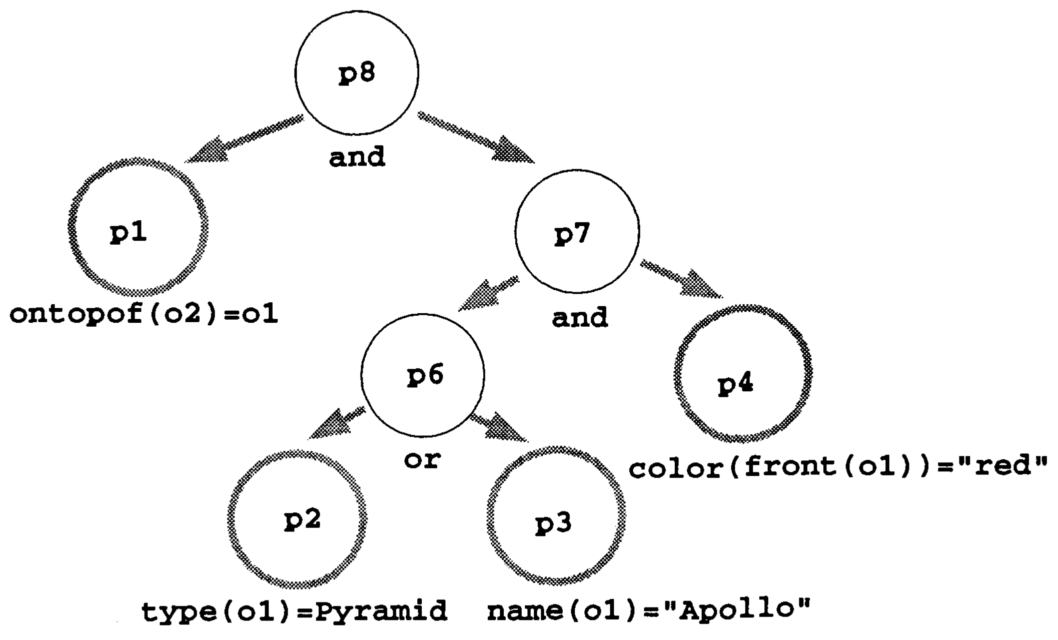


Figure 4.5 **Optimal Parse Tree After Applying Constraints to Spatial Predicate**

$(p2 \vee p3) \wedge p4 \wedge p1$

Thus the following processing plan will bring us better performance.

I. Process the predicate p5 of dependent object o2.

II. Find o1 satisfying condition p4 and (p2 or p3).

III. From o1 in II find the subset satisfying p1.

More contracts are attached to spatial predicate On_Top_Of(o2)=o1, i.e. the size of o1 is reduced before entering the nested loop in function On_Top_Of(o1, o2).

The **optimization rule** when spatial predicates are joined with non-spatial ones is: always try to apply as many as possible non-spatial predicates to constrain the spatial one.

4.3.2 Applying Constraint When OR Operations Are Present

In the last example, there is an **OR** operation in the query, but it has no effect on the spatial predicate. Let's look at the following example, and try to find a way to optimize it.

Example 4.3:

select o1

where

(On_Top_Of(o2)=o1 and type(o1)=Pyramid or color(front(o1))="red")

and name(o1)="Apollo" and

color(top(o2))="green"

go

This query is to find those blocks named "Apollo" such that it is either a pyramid on top of a block with a green top or a block with a red front block.

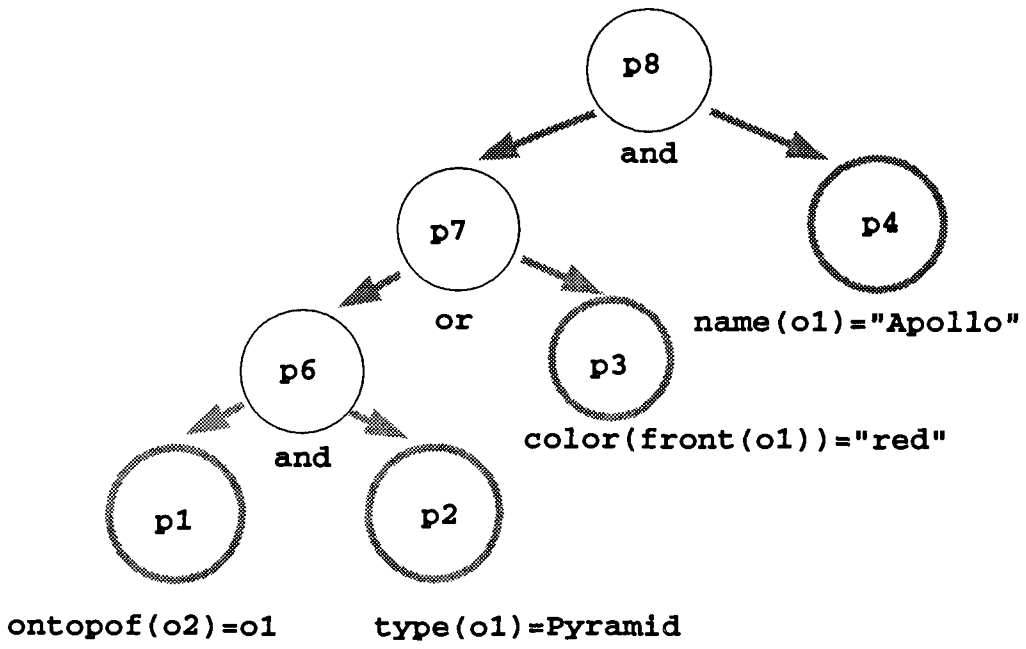


Figure 4.6 **Parse Tree with *OR* Operation Obstacle Applying Constraints**

The logical formula for this query:

$(p1 \wedge p2 \vee p3) \wedge p4$

The processing plan should be:

I. Process the predicate p5 of dependent object o2.

II. Find all the o1 satisfying p2, and from that obtain the subset satisfying p1.

III. Union o1 from II with those satisfying p3.

IV. Intersect o1 from III with those satisfying p4.

The **OR** operation causes us some trouble in applying the constraint name(o1) = "Apollo" to spatial predicate On_Top_Of(o2)=o1. However, we still can apply the constraints using logic computation, and the parse tree will become like this:

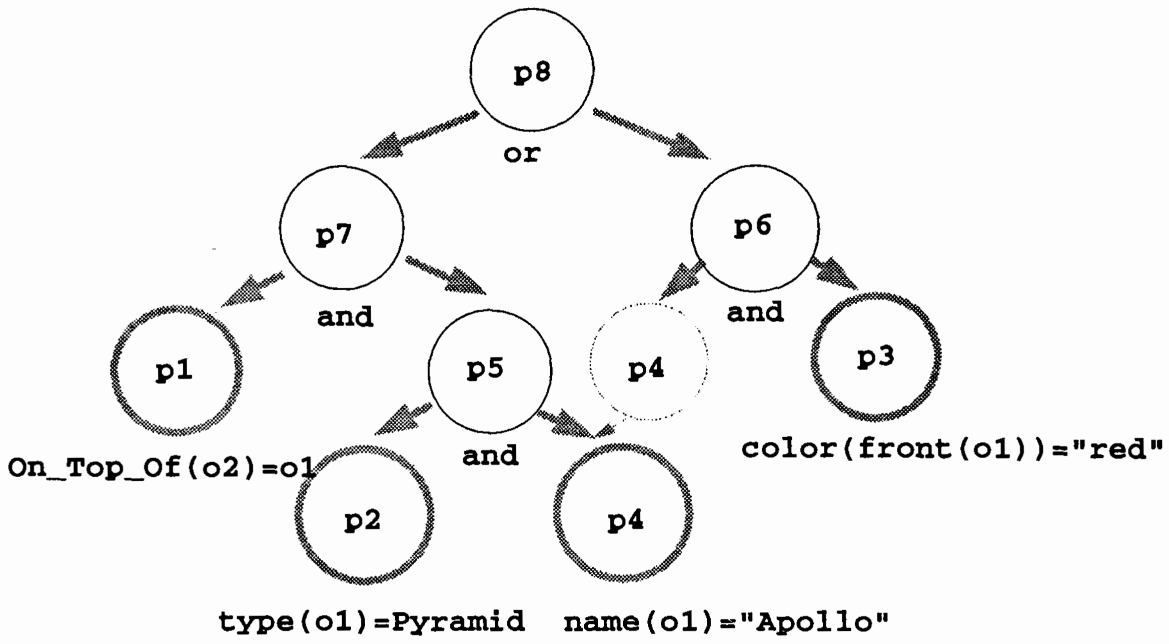


Figure 4.7 **Optimal Parse Tree After Applying Constraints Through *OR* Operation**

The new formula will be:

$$(p1 \wedge p2 \wedge p4) \vee (p3 \wedge p4)$$

There are actually two steps in applying constraint name(o1)="Apollo":

I. Rotate the tree about the AND node to raise the OR node one level up. The left child of the new AND node will be a virtual node, which indicates a linkage to the original sub-branch of the conjunction. In our example this would correspond to p4.

II. Using the same method as in the last example, without an OR operator, to apply constraint, as shown in Figure 4.4.

The processing plan here is quite straightforward:

I. Process the predicate p5 of dependent object o2.

II. Find all the o1 satisfying p2 and p4, and from that obtain the subset satisfying p1.

III. Union o1 from II with those satisfying p3 and p4.

IV. Intersect o1 from II with o1 from III to get the final result

The **optimization rule** is: when spatial predicates are joined with an OR operation, always try the conjunction form (logical).

4.3.3 Propagation of Constraints Through Spatial Predicates

There are times we will have too many spatial predicates, and few or even no non-spatial predicates present. How could we still apply constraints to the query processing?

There are normally three methods to process the AND operation "On_Top_Of(o2)=o1 and On_Bottom_Of(o3)=o1":

1. Parallel (intersection method)

Find the blocks which satisfy On_Top_Of and On_Bottom_Of respectively, and intersect these two sets of results to get the final query result.

```
foreach a_block in BlockWorld do
  foreach block2 in o2 do
    if a_block is on top of block2 {
      save a_block to temp_set1;
      break;
    }
foreach a_block in BlockWorld do
  foreach block3 in o3 do
    if a_block is on bottom of block3 {
            save a_block in temp_set2;
            break;
        }
result_set = temp_set1 & temp_set2;
```

2. Sequential

First find the blocks satisfying On_Top_Of, then in the result set, start testing the On_Bottom_Of predicate to get the final query result.

```
foreach a_block in BlockWorld do
    foreach block2 in o2 do
        if a_block is on top of block2 {
            save a_block in temp_set;
            break;
        }
foreach a_block in temp_set do
    foreach block3 in o3 do
        if a_block is on bottom of block3 {
            save a_block in result_set;
            break;
        }
```

3. Nested

For all the blocks test if it satisfies the On_Top_Of predicate, if so, go on and test the On_Bottom_Of for this object, otherwise, pick the next block to start again.

```
foreach a_block in BlockWorld do
    foreach block2 in o2 do
```

```
if a_block is on top of block2 {
  foreach block3 in o3 do
    if a_block is on bottom of block3 {
      save a_block in result_set;
      break;
    }
  break;
}
```

Let's examine these processing strategies through a concrete example.

Example 4.4:

select o1

where

On_Top_Of(o2)=o1 and On_Bottom_Of(o3)=o1 and On_Left_Of(o4)=o1 and

color(top(o2))="green" and

name(o3)="Apollo" and

type(o4)=Pyramid

go

This query is to find the blocks which are on top of blocks with green tops and below blocks with the name of "Apollo" block and to the left of a pyramid.

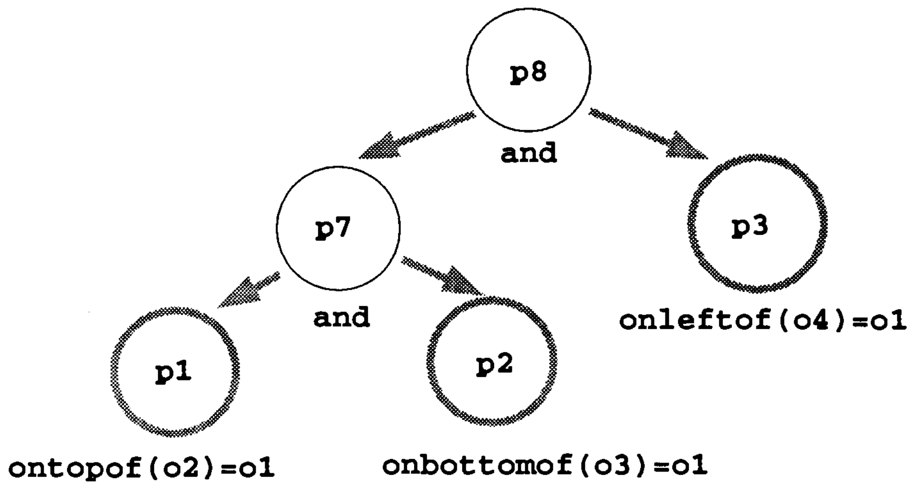


Figure 4.8 **Parse Tree with More Than One Spatial Predicate**

Parallel Method:

I. Process the predicates p4, p5 and p6 of dependent object o2, o3 and o4, respectively.

II. Find those objects satisfying p1.

III. Find those satisfying p2.

IV. Find those satisfying p3.

V. o1 should be the intersection with II, III and IV.

In this strategy, the processing time will be at least doubled if we added one more spatial predicates in the query.

Sequential Method:

As we know from the query, the three predicates are related (i.e. they are all describing the feature of o1). So, once we finish processing the predicate, we should reduce the size of o1, not the whole set of blocks (i.e. apply the constraint from first predicate to the next). Let's assume through one constraint, the size of the possible result will be reduced by 80%. Using this strategy the total access to Block World (N) will be N*(0.2N) + (0.2N)*(0.2N) + (0.2*0.2N)*(0.2N), which is much lower than that without optimization (0.6N*N). The complexity comparison is O(N*lnN) vs. O(N*N). Thus, we have obtained a better processing plan. It is true that the processing order of these spatial predicates will have a great influence on the performance, but we will not discuss this here. The solution for this more or less depends on the statistical information of the database, which will bring in extra costs for each query processed.

I. Process the dependent object o2, o3, and o4.

II. Find those objects satisfying On_Top_Of(o2)=o1.

III. From II find the subset of blocks satisfying On_Bottom_Of(o3)=o1.

IV. From III find the subset of blocks satisfying On_Left_Of(o4)=o1, and this the final result of o1.

Nested Method:

I. Process the dependent objects o2, o3 and o4.

II. For each a_block in BlockWorld, if a_block is on top of one of o2 & below one of o3 and to the left of one of o4, then put a_block in the result set.

We can also prove the complexity of Nested Method is O(N*lnN), if the mapping is 1:1. But if the mapping is m:1, the Nested method is obviously worse than the Sequential method. In our Block World, the mapping for spatial predicates is 1:1, so these two methods are equivalent in complexity. But due to the reason for keeping consistency with the non-spatial predicate, we choose the Sequential Method for our processing strategy.

	1 predicate	*2 predicates*	*3 predicates*
Parallel	19.25 sec.	39.37 sec.	61.22 sec.
Sequential	19.03 sec.	20.32 sec.	25.12 sec.
Nested	19.01 sec.	21.23 sec.	25.76 sec.

Table 4.3 **Comparison of 3 Optimization Strategies in Query Processing**

4.4 Version Control

When CAD applications were first developed, they were intended to support individual engineers working on distinct projects. Today, as design complexity increases, these applications more and more need to support cooperative work by a number of engineers on the same design. As a result, a requirement has emerged for configuration management facilities that allow earlier states of a design to be recorded, and that allow alternative designs to be simultaneously available. The conventional locking scheme will make you wait for your co-worker to finish, because no dirty read is allowed. But with version control, you and your co-worker could each check out your own version and work on them simultaneously. There might be incompatible changes, and the merging of these versions could be a complicated matter, but versioning is more favorable than locking in some cases, especially in CAD areas. Also with version control, you can easily undo your present committed version, and return to a previous one.

We have applied version control to our Block World, and thus provide a better multi-user environment. For example, as shown in Figure 6.1. Two users are cooperating in building a scenery. They both check out a version in their own workspace, and do the following queries.

user1:

```
move o1 to o2
where
type(o1)=Pyramid,
On_Top_Of(o3)=o2,
type(o3)=Cylinder
go
```

user2:

```
move o1 to o2
where
type(o1)=Frustrum,
On_Top_Of(o3)=o2,
type(o3)=Cuboid
go
```

After these queries, they decide to merge the two separate versions into a new one.

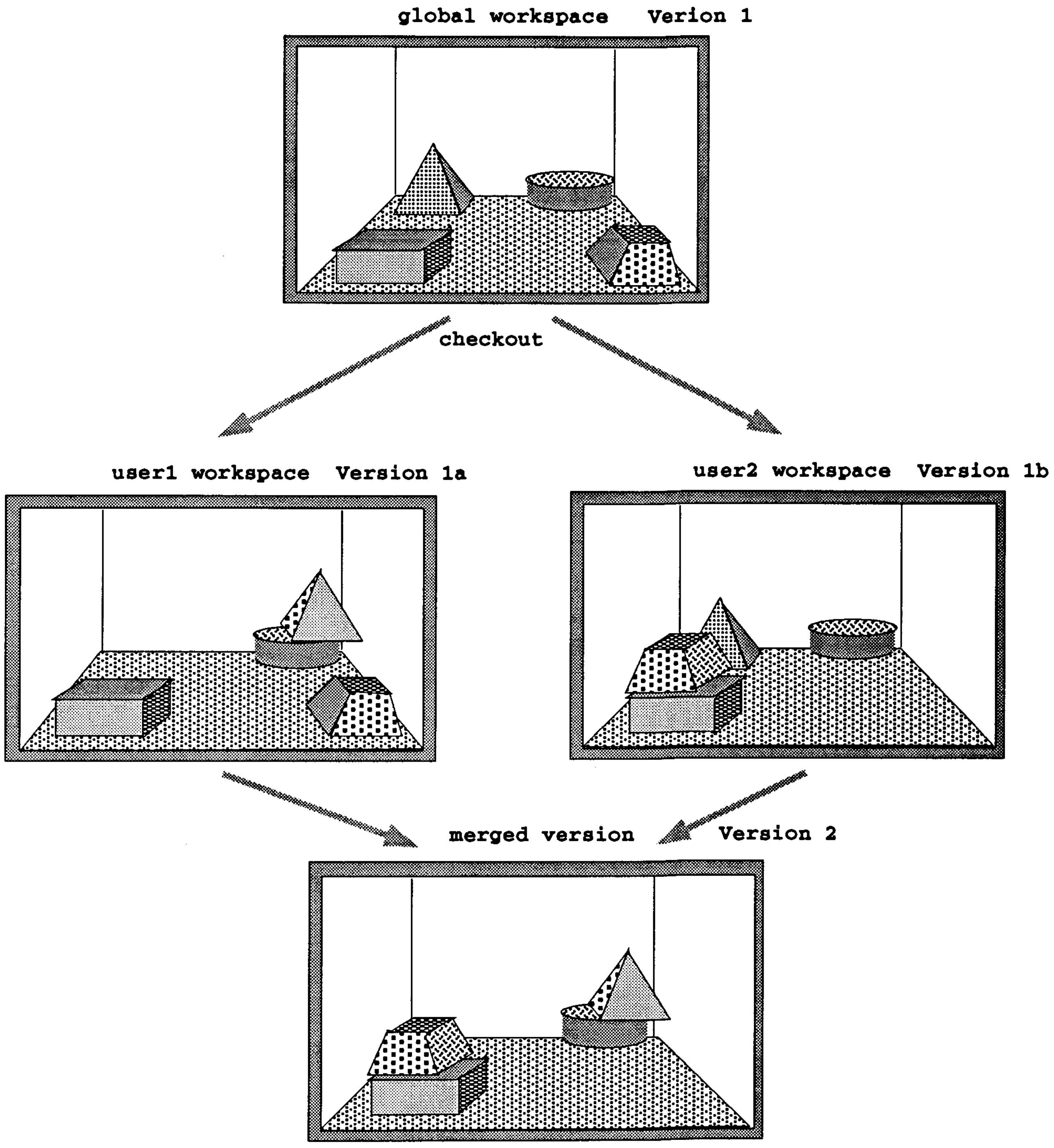

global workspace Verion 1
checkout
user1 workspace Version 1a
user2 workspace Version 1b
merged version
Version 2

Figure 4.9 **Checkout Branch and Merge of Versions**

In ObjectStore, each user can create his own version history inside his workspace, in which everything is private and all the modifications to versions are invisible to other workspaces. But it is still possible to resolve a trans-version object in other versions of the same workspace, or resolve the versioned object in the current version from other workspaces. The first solution will make it possible to traverse the version history of your own workspace, and the second one will be useful for merging the trans-version object (i.e. you may have to find out the status of the merged object by consulting all the current trans-version objects in each branch).

Let's come to the version control of our DSQL. In our system, we will consider the whole set of blocks as a versionable object, as well as each single block. Upon entering the system, there will be a private workspace automatically allocated for each user. For each update query a new version of the scene will be create, which will allow you to backtrace the version history. So we will have to add two additional operations before and after the function:

```
block_config->checkout_branch(user);
query_processing();
block_config->checkin();
```

Where the block_config is a persistent configuration pointer, initialized when the database is created, which serves to group together objects that are to be treated as a unit for the purposes of versioning.

When we decide to undo the current version, and backtrace the version history, we just have to reuse the old versions in the history and forget the current one.

```
void BlockTrace() {
    block_config->predecessor->()->use();
```

```
        using the old version to modify the Block World scenery.
        block_config->forget();
    }
```

The most important task in version control is definitely the merge of version branches. In Block World, we have to create a passive monitoring process to handle the merge, and end users need only to send a signal (UNIX socket) to wake up the merge process. Once the merge is done, all the end users will be notified by a UNIX signal or socket, and they can then carry on work using the new merged version.

```
    void Merge() {
      foreach version branch V (workspace=user_x) {
        workspace::set_current(global);
        config_alt = user_x->resolve(block_config);
        set_alt = config_alt->resolve(extent);
        block_config->merge(config_alt);
        foreach (a_block, extent)
          foreach (block_alt, set_alt)
            if (same_object(a_block, block_alt))
              a_block->Modify(block_alt);
            else
              InsertToGlobal(block_alt);
        workspace::set_current(user_x);
        block_config->forget_branch();
      }
    }
```

Where the InsertToGlobal(block_alt) function constructs an insert query based on **block_alt** and poses the query to DSQL, so that the new block created in version branch will be inserted into global workspace during the merge.

Since the merging of several versions is a background process, the efficiency is not that important. It is easy to imagine that we will encounter more problems using the very sophisticated merging facility provided by the DBMS than some using simple operations. We devise another merge process which only collects all the valid queries[3] executed in each version branch, and reprocess all the queries in the global workspace. Thus we may have a bug-free merging process standing alone. And without the needs for graphical display, the efficiency is not too bad.

[3] The undo queries should not be included.

Chapter 5
Performance Comparison

In this chapter, we will describe some of the variations in implementation of the DSQL on a relational platform. In each case, the performance is compared to other cases as well as an OODBMS implementation.

We will discuss in this chapter two major performance problems of the relational approach in DSQL, **table decomposition** and **impedance mismatch.**

5.1 Implementation Effort in DSQL for Relational DBMS

The relational approach for DSQL is quite different from the object-oriented approach. In the object-oriented approach, important factors, such as object identifier, composite object and data clustering and caching, are the reasons why object-oriented database systems could have much better performance. But another more important reason why a relational database system has very poor performance is the so-called impedance mismatch for the programming data model and database model, while object-oriented DBMS' provide a seamless integration for these two, and hence better performance.

The table decomposition is always a controversial topic. The pure relational approach requires normalization for each table, while in engineering application environments where complex objects are present, the normalization concept is not favoured. In theory, less normalized tables will have better clustering for composite objects, and hence better performance.

The key point in the relational approach is: (1) how to translate the DSQL query into relational SQL and (2) how to make use of relational tables for persistent data handling.

All the spatial and non-spatial predicates will be implemented as relational db-library calls operating on flat tables. You may easily notice the impedance mismatch of the application data model and database data model, i.e. Block and Geometric Objects with hierarchical structure vs. relational flat and normalized tables. While in the object-oriented approach, a unique data representation is present both for the application program (transient) data model and the persistent data model of database.

We have done three kinds of performance comparisons based on Sybase.

I. Pure Relational approach (Strictly Normalized).

II. Less Normalized approach.

III. Relational Approach with PreLoading.

In this chapter we will first introduce two relational approaches, the pure (or strictly normalized) relational and the less normalized one, and describe our DSQL system based on these instead of an object-oriented approach. The purpose for the **Less Normalized** approach is to apply the complex object clustering in the object-oriented concept to the relational concept, expecting better performance. Then we will describe the **PreLoading** approach to integrate the relational system with all the possible object-oriented features, such as object identifier, client caching and data clustering for the composite object. And the most important thing we want to stress, from the performance point of view, is the effect of the "impedance mismatch".

All the comparisons will be done based on schema design, query processing and performance benchmarks. And our interpretation of the performance result will be given for each case. After all these comparison, we will show why we chose an object-oriented DBMS as our basic platform for DSQL implementation.

5.2 Design of Relational Schema

In this section we will describe the schemas both in Pure Relational and Less Normalized approach, and point out the major difference between them.

Let's use the Cuboid as an example to show this difference:

5.2.1 Pure Relational (PR) Approach

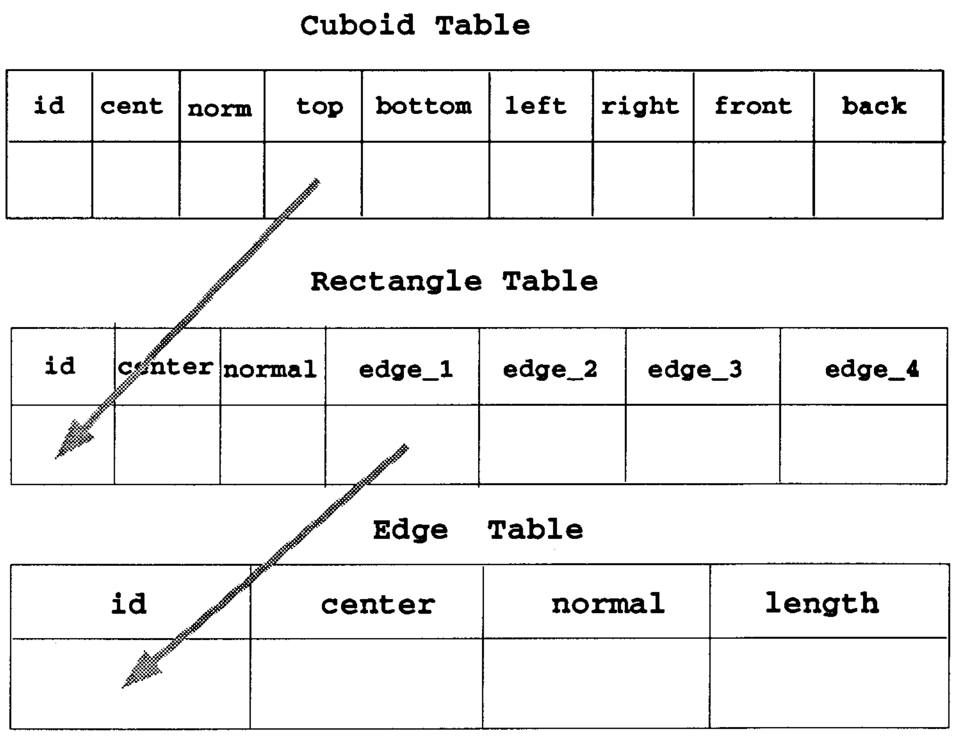


Figure 5.1 **Schema Definition for Pure Relational Concept**

The Cuboid Table is from the Block Level, and the Rectangle and Edge Table are from the Geometric Level. The whole block object is decomposed into three levels of tables for normalization purpose. The field *top* in **Cuboid** links to the field *id* in

Rectangle and the field *edge_1* links to the field id in Edge table. To retrieve all of the cuboid's information, a join operation will be applied to all three tables on related fields. To speed up the join operation, primary indexes are on *id* in each table, and secondary indexes on the join fields, such as *top* and *edge_1*.

5.2.2 Less Normalized (LN) Approach

To obtain high data clustering, table normalization is a main obstacle. We will try to combine some tables to sacrifice normalization for high clustering. The Face and Curve tables are the first targets to be combined. After combining these two tables, we will have two levels of tables for Block and Geometric Objects, respectively.

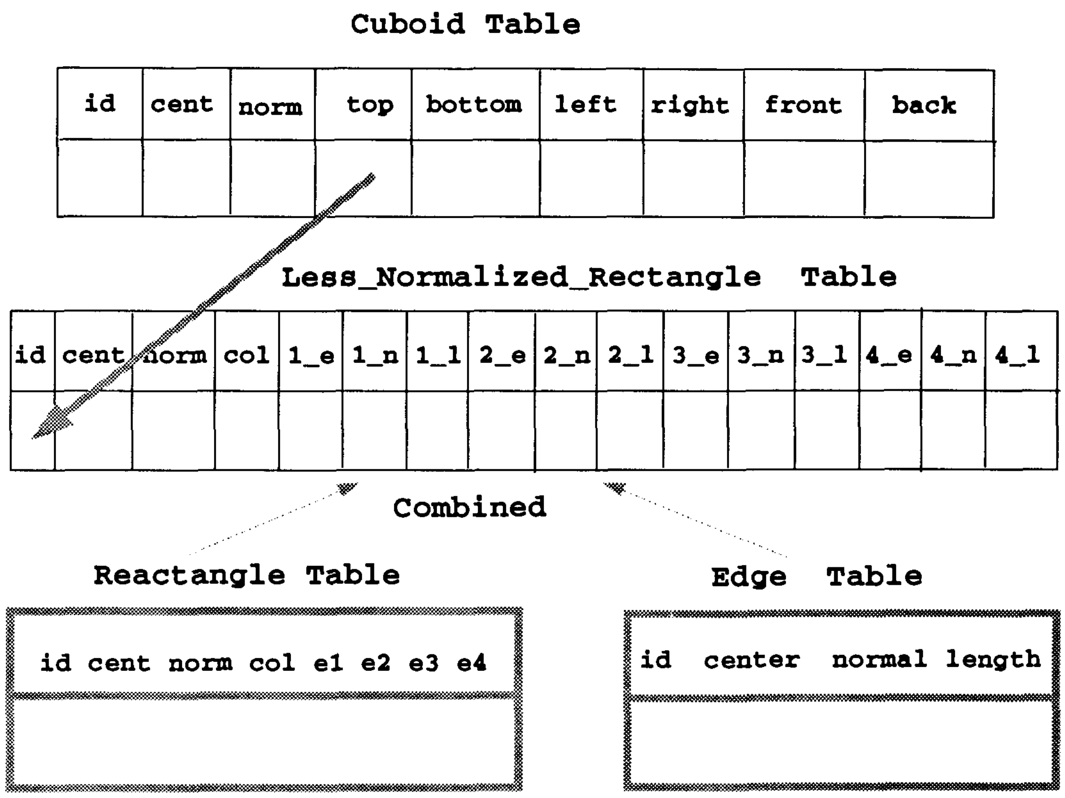


Figure 5.2 **Schema Definition for Less Normalized Relational Approach**

In this approach, we combine the two tables in the Geometric Level into one, and obtain better data clustering, which is closer to the composite object clustering in the object-oriented model. The retrieval for the whole block object will have fewer tables to be joined, and possibly better performance.

5.3 DSQL Query Processing

The general DSQL query processing steps are: query parsing with YACC, generating a processing plan, mapping the plan into database function calls and sending back the result. The difference for each approach is mainly from the execution of database function calls, including spatial and non-spatial predicates processing. Let's look at the difference in PR and LN approaches.

Pure Relational and Less Normalized Approach are much like the same in query processing, except they are different in a Sybase SQL query to retrieve face information.

5.3.1 PR (Normalized) Approach

As we have seen in Chapter 3, the query processing plan will be mapped into the function implementer built on top of the DBMS. In object-oriented database systems, all the spatial and non-spatial predicates implemented as member functions for each class are due to its unique data representation. But in a relational approach things become much more complicated.

Each predicate processed should have the same basic steps:

I. Construct a corresponding SQL query and pose it to the database server (remote).

II. Wait for the SQL result.

III. Bind the SQL result into the local data structure.

The non-spatial processing is relatively simple, here we only show the SQL query to the Sybase server. For performance considerations, a stored procedure is used to save query parsing and compiling

Ex. 5.1: Find all the cuboids with a given colour top.

```
create procedure TopColor @col varchar(10) as
select b.id from cuboid b, rectangle f
where
b.top = f.id and f.color = col
```

go

Ex. 5.2: Find all the cuboids with a given name.

```
create procedure Name @n varchar(10) as
select id from cuboid where name = n
go
```

Because of the incompleteness of SQL, we could not count on Sybase to process a complicated predicate like On_Top_Of. A client program is involved in such processing.

To process a spatial query On_Top_Of(o1, o2), which is basically a join operation. We first retrieve, from Sybase, the bottom rectangle face of block **block_o1** in set o1, then for each block **block_o2** in set o2 get its top rectangle face from Sybase. If this pair of top and bottom faces are overlapped and attached to each other, then we could say **block_o1** is on top of **block_o2**, and store the **block_o1** in the result set. After scanning all the blocks in set o1, we could get a subset of blocks, each on top of at least one block in set o2. Here is the algorithm.

```
On_Top_Of (o1, o2) {
  Lookup the info of o1 and o2 in symbol table;
  for each topBlock in set of o1 {
    Retrieve the face bottom = GetBoundaryFace (topBlock, "Bottom");
    for each bottomBlock in set of o2 {
      Retrieve the face top = GetBoundaryFace (bottomBlock, "Top");
      if face bottom is overlapped with face top
        record this match, topBlock is selected;
    }
  }
}
```

```
GetBoundaryFace (BlockID, faceSide) {
  Based on face side and block type construct the SQL;
  pose the SQL query to the Sybase server;
  bind the SQL result to local data structure;
  return the new face;
}
```

The above functions are the same for both the Pure Relational and the Less Normalized approach, the only differences are in the SQL query.

A join is required to retrieve block faces (e.g. to get one top face we will need to join the rectangle table and the edge table):

Ex. 5.3: retrieve a top face of cuboid.

```
create procedure CuboidTop @block_id as
declare @top_id int
select @top_id = top from Cuboid where id = @block_id
select * from quadplane f, edgecircle e
where f.id=@top_id and e.id = f.e1
..... (repeat for other three curves)
go
```

The reason why we use 4 separate SQL statements instead of one with an OR operator is that the OR operation will be 100 times worse than 4 single SQL statements, due to the features of the query optimizer in Sybase.

5.3.2 LN (Less Normalized) Approach

In the Less Normalized Approach, since we have to join the rectangle and edge table statically (in a table structure), no join operation is required to get a top face.

Ex. 5.4: Retrieve a top face of a cuboid in LN approach.

```
create procedure LessNoramlizedCuboidTop @block_id as
declare @top_id int
select @top_id = top from Cuboid where id = @block_id
select * from less_normalized_quadplane
where id=@top_id
go
```

In the new rectangle table, all the information for one face will be stored in one tuple, and definitely in a page. While in the normalized one, the same amount of information is separated into two tables, and possibly the query for the rectangle face will require more disk I/O.

5.4 Performance Comparison

In this section we will do the following comparisons:

I. Spatial Predicate comparison in which a single type and assorted types of blocks are investigated as well as disk I/O.

II. Non-spatial Predicate.

Let's first get a overview of all the DSQL operations.

All the query conditions for non-spatial query comparison is:

I. One Non-spatial Predicate: color (top (o1)) = "red".

II. Two Non-spatial Predicates: color (top (o1)) = "red" and color (front (o1)) = "green".

All the query conditions for spatial query comparison is:

I. One Spatial Predicate: On_Top_Of (o2) = o1 and color (front (o2)) = "red".

II. Two Spatial Predicates: On_Top_Of (o2) = o1 and On_Bottom_Of (o3) = o1 and color (front (o2)) = "red" and color (top (o3)) = "green".

	ObjectStore		***Sybase***	
	1 pred	***2 pred***	***1 pred***	***2 pred***
insert	1.84	N/A	1.19	N/A
move	0.58	0.76	2.24	4.80
rotate	0.62	0.79	2.25	4.73
select	0.67	0.78	2.19	4.26

Table 5.4 **Non-spatial Predicate Comparison Based on 500 Cuboids**

	ObjectStore		***Sybase***	
	1 pred	***2 pred***	***1 pred***	***2 pred***
move	18.63	20.96	21.13	28.75
rotate	19.25	20.78	22.32	27.61
select	19.03	20.32	20.20	22.98

Table 5.5 **Spatial Predicate Comparison Based on 500 Cuboids**

Since move, rotate and select are not much different in performance, we will continue the comparison on only the select DSQL.

5.4.1 Spatial Query Comparison

It is well known that cold and warm comparisons sometimes make a big difference, especially when client machine caching is applied. The following results are comparisons between a cold and warm environment.

In the block world, the **Warm** means we first call one function to access all the objects in a database once, then test the spatial and non-spatial queries. In the warm stage, some objects may be cached, some may be swapped out already. The **Cold** means we will execute the spatial and non-spatial queries with no objects in the database cached. The importance for testing both Warm and Cold cases is that they usually make a nontrivial difference.

	100		*200*		*500*		*1000*	
	warm	**cold**	**warm**	**cold**	**warm**	**cold**	**warm**	**cold**
Normalized	104.7	112.7	410.5	439.4	2230.7	2256	8724	8674
Less Normal	310.8	348.2	1728	1807	9098	9170	36891	37268
ObjectStore	1.43	1.96	5.74	7.04	19.67	26.97	72.60	99.24

Table 5.6 **Warm & Cold Comparison (Select With 1 Spatial Predicate on Cuboids)**

Since the last comparison is only on Cuboid, we may want to try assorted blocks to see if it makes a difference. The following example is on 4 kinds of blocks, Cuboid, Cylinder, Frustrum and Sphere.

	100		200		500		1000	
	warm	cold	warm	cold	warm	cold	warm	cold
Normalized	93.6	98.2	301.3	327.4	1628	1655.	8572	8559
Less Normal	268.1	271.3	1171	1225	6286	6317	35120	34921
ObjectStore	1.24	1.68	3.91	4.37	14.26	17.50	64.30	78.23

Table 5.7 **Warm & Cold Comparison (Select with 1 Spatial Predicate on Assorted Blocks)**

Based on the above result, people may be shocked at the result, even that they may expect an OODBMS would be a lot better than a RDBMS in our scenario. But still this is only one part of the story, for a complicated predicate, the OODBMS does amazingly well. But what about the simple query with only non-spatial predicates? We will see the comparison in next topic.

Another surprising result is that the LN Approach is worse off than the PR approach. What's wrong? The normalized table format with many join operations could be better than the less normalized table with no join operation, which is completely unbelievable. We decide to make use of the statistical information in Sybase to investigate the mystery. Following is the timing and physical disk I/O accumulated from each Sybase SQL query.

	100		200		500		1000	
	time	io	time	io	time	io	time	io
Normalized	315.7	1279	1007	11079	5308	89354	20221	375023
Less Normal	187.2	1097	635.1	8947	3427	53729	13684	229443

Table 5.8 **Performance Comparison with Sybase Statistics I/O Turned On**

Another surprising result comes out. It is easy to imagine the performance of the normalized approach to be worse, but it should not be that much different. While this seems to be an incredible result when turning on the statistics I/O in a less normalized one. We will try to explain it later.

5.4.2 Non-Spatial Query Comparison

	100		*200*		*500*		*1000*	
	warm	**cold**	**warm**	**cold**	**warm**	**cold**	**warm**	**cold**
Normalized	0.091	0.12	0.32	0.49	1.31	2.13	2.79	4.41
Less Normal	0.13	0.17	0.46	0.58	1.79	2.98	4.73	6.17
ObjectStore	0.072	0.42	0.31	0.89	1.21	4.47	2.45	9.23

Table 5.9 **Warm & Cold Comparison (Select with 1 Non-spatial Predicate on Cuboid)**

From this result, we could get to know where the object-oriented database system is best suited. For the simple query or some other business application, object-oriented databases may not be promising.

5.5 Discussion on Performance Comparison

Object-oriented database systems are best suited for our DSQL for the following reasons:

5.5.1 Relational Key vs. Object Identifier

As we mentioned before, two important features of the object-oriented DBMS are the notions of a complex object and an object identifier.

On the one hand, each object has its own identity represented by a unique OID. On the other hand, the notion of complex object allows the aggregation of heterogeneous objects as a single unit. [3]

In the DSQL object-oriented approach, the complex Block object has its sub-objects (attributes) Boundary Face from Geometric Class, and is linked together as a graph structure. The references to face object are made explicitly through face OID links. Further, object-oriented systems, like ObjectStore, have been developed largely independently of any consideration for very large databases (i.e. they assume that all objects reside in a large virtual memory). This means that object identifiers have been used as the sole means of specifying desired objects, which allows the dereference to Block and Geometric object with great ease and amazing performance.

A major advantage of object-oriented DBMS' is that many relational join operations can be prevented when retrieving complex objects; instead of joining tuples from different relations based on some attribute values, objects are simply accessed through their OIDs.

In the relational approach of DSQL, when we want to process a spatial query, the only persistent identifier for Block is block id (i.e. key in relational tables). Given a blockID, whenever we call GetBoundaryFace(blockID), we have to do 1 select and 4 join operations which is the most expensive SQL operation. While in the object-oriented approach, given a block object identifier, with pointer swizzling in ObjectStore, we could get its address in virtual memory at once. Traversing through the Block Object and Geometric Object hierarchical tree, the information of the face boundary will be obtained at maximum speed. No join operation is needed, only navigation of a class hierarchy. There is only one operation like this, the difference may be trivial, but the spatial predicate is just a sort of join and this operation will be repeated O(N*N) times. So it is not difficult to explain why an object-oriented database system processing a complicated DSQL is more than 100 time faster than a relational one.

The accessing of a Block object through the block key, a remote SQL query to Sybase, and client side data binding emphasize the problem of impedance mismatch in a relational approach. In comparison, the object access in ObjectStore is so natural and efficient just because of its close integration of database schema and programming data model, shown in Chapter 2.

5.5.2 Client Caching and Server Caching

People may raise a question immediately after we said the join operation is one of the major reasons why the relational approach is so slow. What about data caching in a Sybase server (10M buffer size), —it is highly possible that join operations are done in the buffer (memory), if the join attributes are properly indexed?

It is true that the relational database system is trying to do all the joins in the buffer for high performance consideration, and Sybase really does a very nice job in query optimization. Sybase is a typical client-server architecture, and all the client performance depends heavily on the server's efficiency. This is a so-called centralized processing, while the client program is relatively small, requiring very little memory. But one thing we should notice, the relational database server is developed for a multi-user environment. Even though its resources are considerably large, each user has a strict limit.

Again, client caching for the relational approach is not suitable in DSQL spatial predicate processing. Sybase could also provide a client side caching (buffering), when a large number of resulting tuples are returned from the server, the client could have a buffer pool for those resulting tuples, and make use of the data in the client buffer pool. High performance is gained accordingly. But our case is completely different, what we have is a small piece of returning data and extremely high frequency usage for each SQL (i.e. this SQL is executed thousands of times in one spatial query processing). Thus Sybase's client buffering could not bring us high performance.

Let's go back to ObjectStore when client caching is used, the database server is only involved whenever a page fault occurred in the client program. ObjectStore is a typical CLIENT-server architecture, and the server's efficiency is no longer a critical factor, as the client machine is normally workstation. This is basically decentralized processing, the client program is quite large and requires more memory. When data is fetched into memory, it is cached in the client machine (local workstation). Since workstations are mostly for single-users, the application buffer size for client caching is therefore larger than server caching, even the total buffer size of the server is definitely much larger.

So with DSQL, client caching is also one of the important reasons for ObjectStore outperforming Sybase. We can see from the following table how many times a boundary is crossed in spatial predicate query processing.

	100	*200*	*500*	*1000*
# crossing boundary	1667	6668	41675	166667

Table 5.10 **Crossing Boundary in Relational Database System**

5.5.3 Data Clustering

Object-Oriented vs. Relational From the efficiency point of view, one of the most important methods is to use the complex object as a unit of clustering in the database, so that a large collection of related objects may be retrieved efficiently from the database. The typical operations performed on the hierarchy of complex objects are the navigation through links and the retrieval of the ancestors/descendants of a given node. [3]

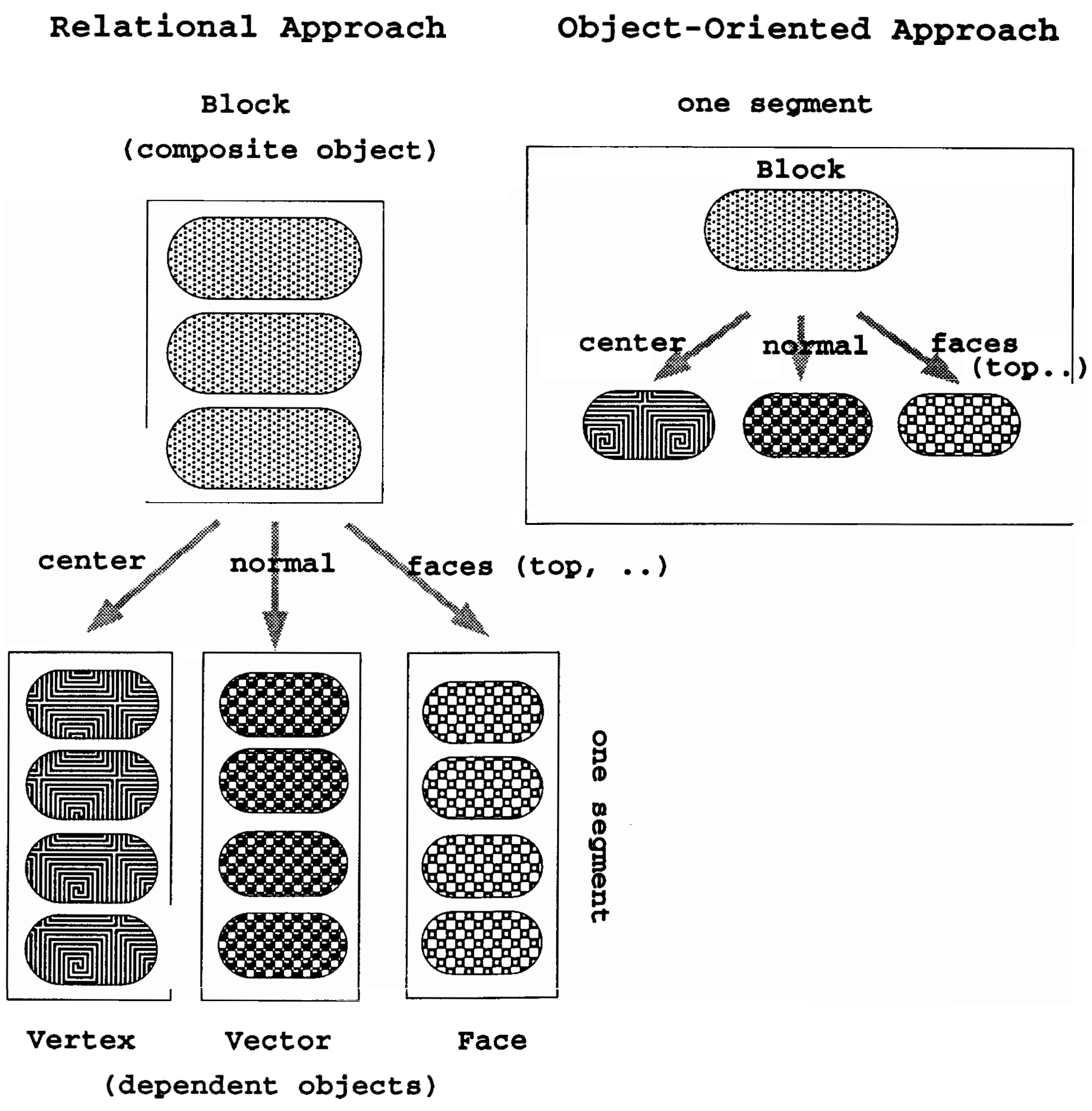


Figure 5.3 **Composite Object Clustering**

The boundary representation of geometric objects is clearly a hierarchical data structure, so for normalization purposes several levels of tables should be created, in Sybase, sacrificing the benefit of data clustering. While in an OODB (ObjectStore), all the data members objects are clustered during the object construction, and hopefully one disk I/O could retrieve the complex object if it could fit into one page. When a composite object's (cuboid) instance is created, all dependent objects (e.g. faces[4] and centre) are created accordingly and clustered in the disk with the cuboid's other data members (e.g. name). Due to its small size, all the objects, composite and dependent, could be stored in one page. So the retrieval will be done in one disk I/O, while the non-clustered approach will require much more I/O.

Normalized vs. Less Normalized In the **Pure Relational** approach of DSQL, we have the maximum possible clustering in each table for performance reasons, but the decomposition of three levels of tables for normalization purposes, stop us from any more performance improvements. That is the reason why we have come up with another relational approach, **Less Normalized,** simply by reducing the degree of normalization (i.e. combine the Face table and Curve one into a larger table and hence better clustering). But the result is quite unexpected. Why?

Sybase's query optimizer normally will assign more resources to the stored procedure in a normalized approach, since join operations are involved. However, the stored procedure in the less normalized approach has no join operation at all, and naturally a Sybase optimizer will consider it to be a non-critical query, and hence assign less resources to it. That is the reason why we could not get better performance in the less normalized approach, as seen in Table 5.9.

[4] Which is also a composite object.

When we turn on the Sybase statistics I/O, a completely different result comes out. The problem, we suspect, is related to server caching. As we mentioned before, the server is designed for a multi-user environment, and each client process has a limit of the server resources. Even though there is only one client process running, the server will still be unwilling to assign more resources to it than it limits. Due to different requirements, analyzed by the query optimizer, the stored procedure in the normalized approach will definitely obtain more resources (e.g. buffer) than that in the less normalized one, and hence better performance. But when the statistics I/O of Sybase is turned on, for general purpose, a fixed amount of resource will be assigned to the SQL query. Since the first stored procedure is much more complicated, 5 select statements, than the second, the statistics process for it will have a heavier load and may even use up some of the resources assigned to the first stored procedure. So the first stored procedure will be delayed by less resources and also more statistics information handling. The second one is different as it may receive more resources than before, thus better performance.

Irrespective of the actual outcome of the investigation, one thing is quite clear. The query optimization techniques used by a relational database system have their own limitation. In the absence of application specific information, it cannot produce an optimal query processing plan. Secondly, as the server caters to a large number of clients, with a high variation of workload, its objective of design is to provide the largest possible throughput for all the transactions, not the performance of a single query.

5.5.4 Complexity of Query

The only part when the object-oriented approach is not satisfactory is the cold stage for simple (non-spatial predicate) DSQL processing.

The client-SERVER architecture is best suited for simple on-line SQL processing (e.g. business applications). While the CLIENT-server architecture requires a relatively longer time for the warm up of less powerful client machine. That is why we get worse performance of the DSQL in this scenario. But once the client machine has been warmed up, the performance is greatly improved, and even produce a better performance than a client-SERVER one.

5.6 Relational Approach with PreLoading

In the above section, we have discussed several factors that contribute to the performance advantage of the OODB approach. It is clear that we could not improve the performance of the relational approach by the traditional methods (e.g. pre-join). In this section, we describe an experiment that attempts to narrow the performance gap.

Borrowing the idea of ObjectStore Memory Mapping and Pointer Swizzling, we come up with the idea to use client caching by preloading all the data set into the client machine's virtual memory and handle the object by its virtual memory address instead of the relational key. This is called PreLoading.

In the Block World, we would like to see that given a block ID which is unique, we should be able to link it to an object address inside of virtual memory. Whether this object is in memory or in swap space is clearly beyond our interest, and could be handled by the OS. With the help of preloading, all of these can be accomplished, and hence better performance and more concise coding. Also, in the preloading strategy, data clustering is achieved by constructing the composite object in the consecutive addresses in virtual memory, just like what ObjectStore does in data clustering. The retrieval of one block object may at most require one disk I/O.

5.6.1 Query Processing

The schema for PreLoading is like a hybrid of relational and object-oriented approaches, in a sense that the database server side is still a relational database, while the client side the DSQL is processed in exactly the same as the OODB approach. A virtual object-oriented database view is presented to the DSQL OODB query processor. This virtual DB will provide a block pointer, given a read block command, see Figure 5.4.

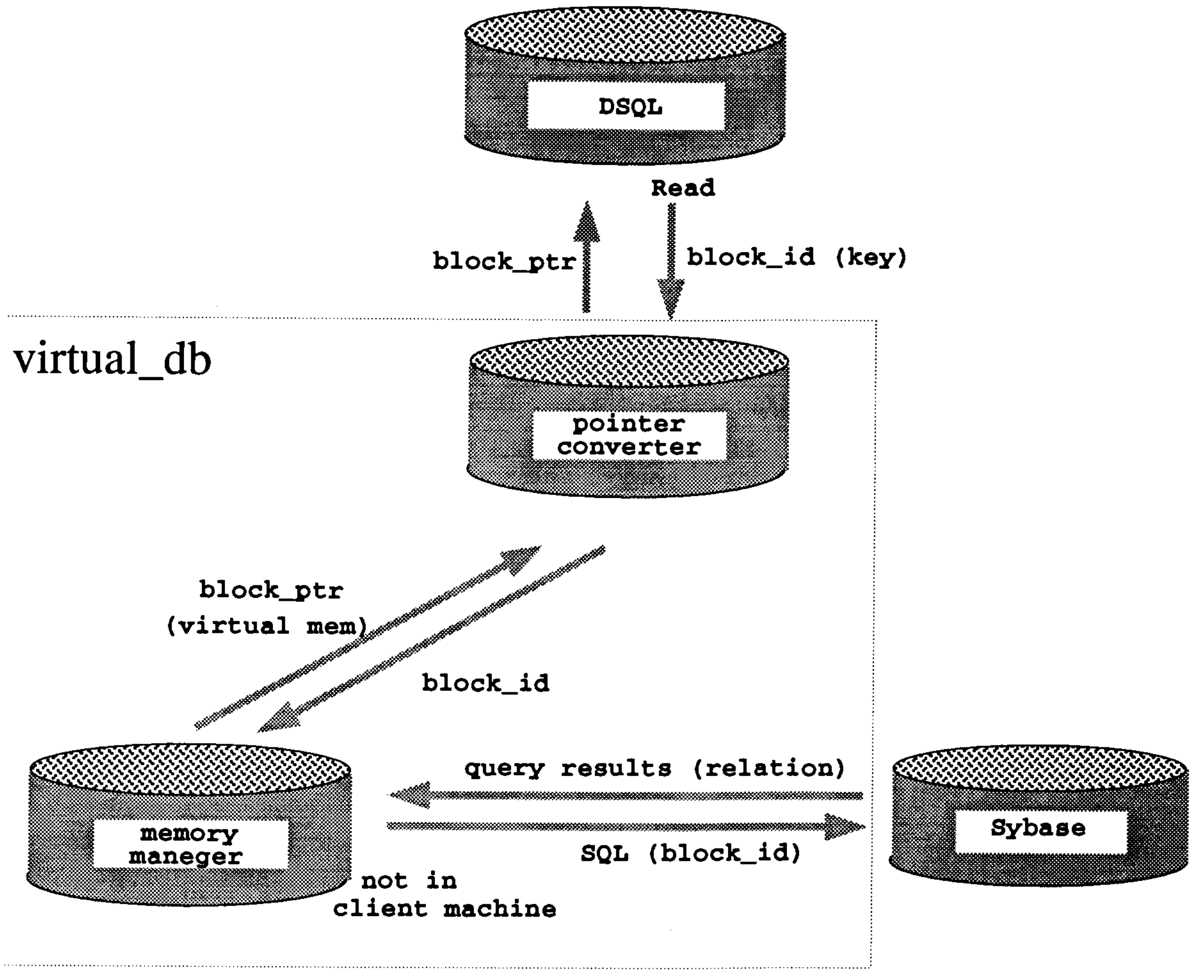


Figure 5.4 **Relational Key Links to Object Pointer in Virtual Memory (read)**

A virtual database system can be defined as following:

```
class virtual_db {
    private:
        PointerConverter pointCvt;
        MemoryManager memoryMgr;
    public:
        /* Provide a block pointer in client machine, given a block key */
        Block* ReadBlock(int block_id);
        /* Update the block info in database from the block info and key */
        void WriteBlock(Block* theBlock, int block_id);
        /* Insert a new block into database and keep track of its pointer */
        void WriteNewBlock(Block* theBlock);
        /* Delete a block in database, and free its space in client machine */
        void DeleteBlock(int block_id);
}
```

The only difference of the PreLoading method to the object-oriented one is just that we have to explicitly call the ReadBlock, WriteBlock, WriteNewBlock and DeleteBlock to do block handling (e.g. if we want to access the name of block, we will do { Block* a_block = ReadBlock (block_id); a_block->Name()... }, not just like in ObjectStore as we could do direct access through a Block pointer.)

As you see in Fig. 5.4, a virtual database server is provided for users to read, write, delete and insert (e.g. read operation will send a block_id (key in Sybase) to virtual_db). The virtual_db sever will invoke the pointer converter to hash for the object pointer in virtual memory. Upon success, return the pointer, otherwise the memory manager invokes db-library calls to pose the SQL (stored procedure) to Sybase to obtain all the necessary information of the block object, which are a set of records, finally allocate space in the virtual memory (heap) and recursively bind information to the block (complex object) and its boundary, finally the pointer converter will maintain the link of the block key to the block pointer of virtual memory in the hash table. Since it is obvious we could not allocate space in virtual memory (swap space and memory) all the time, due to the limitation of swap space, once the swap space has overflowed, first–in–first–out (FIFO) strategy is used for objects replacement. Thus we create a linkage between the block key (persistent) and its transient pointer in virtual memory, avoiding accessing Sybase to retrieve a block all the time.

```
Block* ReadBlock (int block_id) {
    1. Invoke the pointer converter to hash for the block pointer;
    2. If success then return the block pointer in virtual memory;
    3. Upon failure, invoke the memory manger to send SQL to remote Sybase;
    4. Wait for the result of SQL;
    5. Allocate virtual memory space for the composite block object.
    6. Recursively bind the SQL result to block object, including sub-objects;
    7. Invoke the pointer translator to maintain the link (key to VM pointer);
    8. Return the pointer;
}
WriteBlock (Block* theBlock, int block_id) {
    1. Convert the information of theBlock into SQL queries;
    2. Send the SQL to Sybase to update the old block info;
    3. Wait for the SQL to finish;
}
WriteNewBlock (Block* theBlock) {
    1. Convert the information of theBlock into SQL queries;
    2. Send the SQL to Sybase to insert this new block;
    3. Wait for the SQL to finish;
    4. Keep this new link of key to pointer in hash table;
}
DeleteBlock (int block_id) {
    1. Send the SQL to Sybase to delete this block;
    2. Wait for the SQL to finish;
    3. Remove the link of key to pointer in hash table;
    4. Deallocate the space of this block in client machine;
}
```

5.6.2 Performance Comparison

The performance comparisons for the preloading strategy are:

	100	*200*	*500*	*1000*
PreLoading	21.5	43.7	114.34	220.8
ObjectStore	7.41	11.37	30.05	67.21

Table 5.11 **Performance Comparison During Initialization**

	ObjectStore		*Sybase*	
	1 pred	*2 pred*	*1 pred*	*2 pred*
insert	1.84	N/A	0.83	N/A
move	0.58	0.76	1.37	1.42
rotate	0.62	0.79	1.35	1.41
select	0.67	0.78	1.31	1.35

Table 5.12 **Non-spatial Predicate Comparison Based on 500 Cuboids**

	ObjectStore		*Sybase*	
	1 pred	*2 pred*	*1 pred*	*2 pred*
move	18.63	20.96	17.98	20.19
rotate	19.25	20.78	18.94	21.15
select	19.03	20.32	19.30	22.01

Table 5.13 **Spatial Predicate Comparison Based on 500 Cuboids**

5.6.3 Discussion

The PreLoading method seems to solve the problem of impedance mismatch between database schema and programming data model, and thus we could apply all the efficient methods to data manipulation. Client Caching, OID Accessing, Data Clustering built on top of a relational database could bring us an amazing improvement of performance with a reasonable cost. This shows how much the object-oriented concepts are important to engineering applications.

Our Preloading Strategy for linking block_id (key) to block pointer in virtual memory also could be considered as an emulation of memory mapped I/O architecture. We are trying to preload all the objects in the virtual memory at once and upon a page fault the operating system will take charge of swapping in the page we need. But our strategy still could match with the real memory mapped architecture for two reasons:

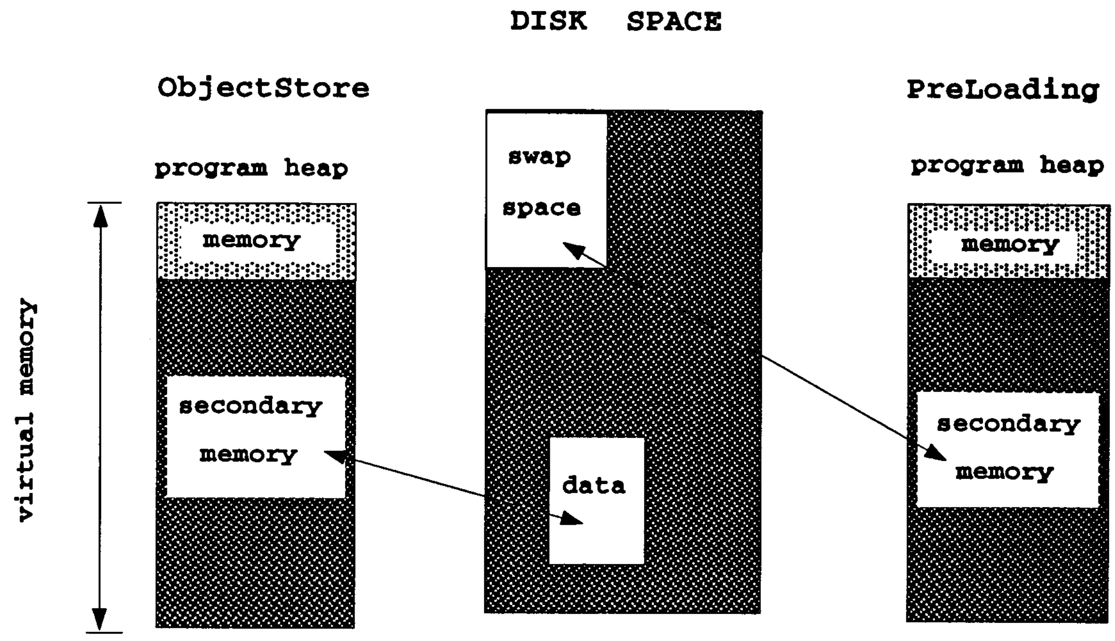


Figure 5.5 **Indirect Mapping Based vs. Memory Mapped**

Initialization of the DSQL system:

I. Initialization Cost: Create all the blocks in the virtual memory. During initialization, all the blocks read in from the data space on disk will be placed in virtual memory, which includes real memory and swap space, so an extra disk copy (from data space to swap space) is unavoidable. It is obvious that the limit for virtual memory is the swap space size.

II. Swap Space Size: For PreLoading, obviously the limit for virtual memory is the swap space size, while no swap space is needed for ObjectStore. An object is swapped into the memory directly from the data space only when a page fault occurs, which means a dereference to the object pointer in memory was failed and the ObjectStore server is awakened to swap in the page with the specific object.

So the size of the virtual memory = real memory + data size, and therefore the virtual memory should be considered limitless.

On the other hand, PreLoading does have some advantages over the OODB approach:

I. PreLoading: The swap space is either local or remote. When there is local swap space available, the object could be retrieved locally without any network communication delay.

II. ObjectStore: The data space is either local or remote. When data is stored on a remote disk, there is an overhead of network communication for object retrieval. Further, there is an overhead of context switching in awakening the OS server when a page fault occurs.

For the above reasons, we could understand why some results of the PreLoading method are a little better than ObjectStore. As seen in Table 5.13. Further, the data operations of ObjectStore are always done remotely, while for the two other relational approaches with PreLoading the data operations could be done in the local swap space without any network communication overhead.

However, the Pre-Loading method could not solve the problems of data recovery and concurrency control, and to develop a whole database system with data recovery and concurrency control is too extra work. So OODBMS is best suited for this kind of application with impressive performance and ease of coding.

5.6.4 Summary

From each of the 4 kinds of implementation approaches for DSQL, it is observed how important the implementation using an object-oriented concept is.

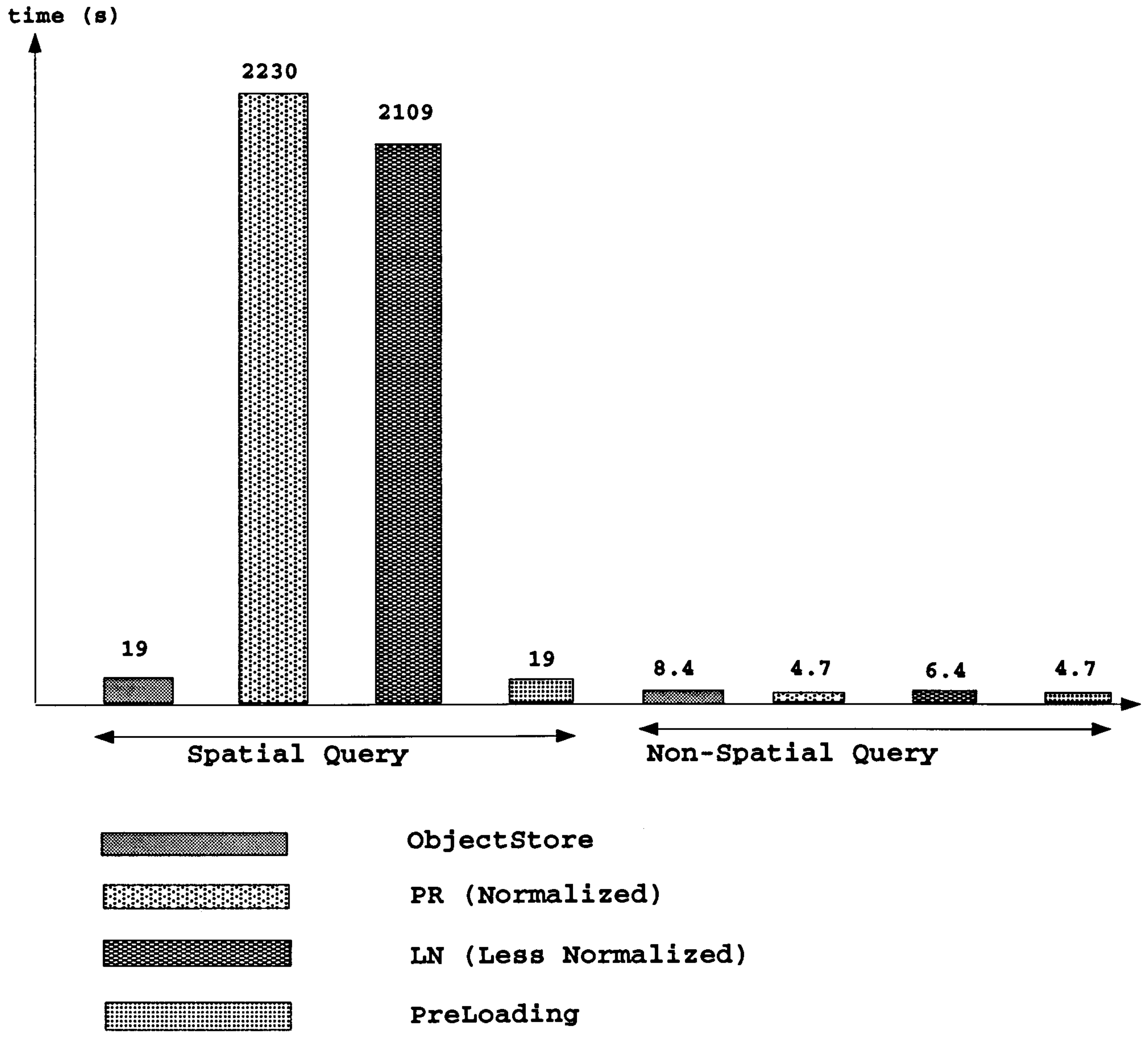


Figure 5.6 **Performance Comparison of 4 Approaches in DSQL**

Chapter 6
Conclusions

Much work has been done in how to process an ad hoc on-line query efficiently, and how to apply the version control to engineering applications. The query processor and optimizer were carefully designed to achieve a better performance. The performance results show that the customized query optimization strategies could bring much better performance than the generic one supported by the DBMS. It is easily seen from the DSQL implementation efforts, an ASQL query processor and optimizer are affordable.

A passive monitoring process will provide version management, especially during the merging operations. We have built a system with an X Window interface along with a DSQL interface to Block World, HOOPS (X Window based graphic package) is used for the 3–D display.

With the implementation effort on DSQL, the benchmark obtain from a more meaningful environment for performance comparison which make use of the spatial query language. Data clustering and memory mapped architecture are important in the performance comparison benchmark, which closely matches the two most important notions of an object-oriented approach — composite object and object identifier, while none of these are present in the pure relational approach. Comparisons on the absence and presence of data clustering or memory mapped feature will give another proof on the importance of these two features to engineering applications.

After finishing the performance comparison, it is easily seen what the major advantages and disadvantages are in using OODBMS' and relational DBMS' for low level data management. The impedance mismatch is the real bottleneck in the relational approach, and the normalization is not always a factor for poor performance in engineering applications. Further, we have successfully applied most the object-oriented features to relational databases using PreLoading, and gained almost the same performance result as in ObjectStore. But since Pre-Loading method could not solve all the problems, such as data recovery and concurrency control, we still consider OODBMS is a better choice of platform to implement an ASQL.

From all these performance comparisons, it is obvious that in building a GIS on an OODB is much better than building on a relational one, with regards to performance and coding[5]. There are many important reasons why an OODBMS (ObjectStore) is better suited for our GIS than a RDBMS (Sybase) might be. Speaking generally, the model supported by ObjectStore often matches more closely to our data model. This means that there is less overhead for joining things together which may have been split up due to normalization. Further, there is much less translation between the data on disk and the host language's data model when ObjectStore is used.

Using this strategy, the preprocessing for the DSQL will be constantly less than 2 seconds, which only occupies 5–10% of the whole query[6] processing time. Thus, we could meet the performance criteria for an ad hoc query's on-line processing, and provide a unique query processing strategy for both interactive and embedded usages.

From the development of the DSQL query processor, using a Bi-Level data model and an OODBMS, the development of an ASQL query processor is easy and could bring the maximum benefits.

---

5 Even though the power of client machine really does matter in ObjectStore.

6 A query with reasonable complexity, i.e. with spatial predicate(s).

In future research, a R-tree index will be created in the spatial predicate processing, and the investigation will be done in how indexing affects the query performance and object-oriented and relational comparison. Since the impedance mismatch is the bottleneck of query processing in relational systems, another interesting issue would be the crossing boundary in OODB system. More experiments will be done in how to improve the query processing in OODB systems.

Reference

[1] Aref, W.G. and H. Samet

Optimization Strategies for Spatial Query Processing.

Proceedings of the 17th international conference on VLDB, Sept, 1991.

[2] Cattell, R.G.G.

Object Data Management — Object-Oriented and Extended Relational Database Systems.

Addison–Wesley Publishing Company, 1991.

[3] Cheng, J.R. and A.R. Hurson

Effective Clustering of Complex Objects in Object-Oriented Databases.

ACM SIGMOD 91's Conference Proceedings, Vol 20, June 1991.

[4] Choi, A.Y.L. and W.S. Luk

A Bi-Level Object-Oriented Data Model for Geographic Information Systems.

IEEE COMPSAC, Chicago, October, 1990

[5] DeWitt, D. J.

The Wisconsin Benchmark: Past, Present, and Future.

The Benchmark Handbook — for database and transaction processing systems.

Morgan Kaufmann Publishers 1991.

[6] Dual, J. and C. Damon

A Performance Comparison of Object and Relational Databases Using the Sun Benchmark.

OOPLSLA '88 Conference Proceedings, SigPlan Notices, Volume 23, Number 11, November 1988.

[7] Kim, W.

Object-Oriented Databases: Definition and Research Directions

IEEE Transactions on Knowledge and Data Engineering. Vol 2. No. 3. September 1990.

[8] Lamb, C., G. Landis, J. Orenstein, and D. Weinreb

The ObjectStore Database System.

ACM Communication. Vol 34, No. 10, October 1991.

[9] Luk, W.S. and A.Y.L. Choi

Dynamic Spatial Query Language: A Customized Query Language for Object-Oriented Database Systems.

IEEE COMPSAC, Tokyo, Semptember, 1991.

[10] Moss, J.E.B.

Object-Oriented as Catalyst for Language-Database Intergration.

Technical Report MIT-LCS-260, PhD. Thesis, MIT.

[11] Orenstein J., S. Haradhvala, B. Margulies and D. Sakahara

Query Processing In the ObjectStore Database System.

ACM SIGMOD 92's Conference Proceedings, Volume 21, June 1992.

[12] Rosenberg, J.B.

Geographical Data Structure Compared: A Study of Data Structures Supporting Region Queries.

IEEE Transactions on Computer-Aided Design, 4, 1, January 1985

[13] Samet, H.

The Design and Analysis of Spatial Data Structure.

Addison-Wesley Publishing Company, 1990

[14] Snodgrass, R.
The Temporal Query Language TQuel.
ACM Transactions on Database Syetms, Vol. 12. No. 2. June 1987.

[15] Stonebraker, M. and L.A. Rowe
The POSTGRES Data Model.
Proceeding of the ACM SIGMOD Conference, Washington D.C. 1986.

[16] Strousstrup, B.
The C++ Programming Language.
Addison-Wesley Publishing Company, 1986.

[17] Winslett, M. and S. Chu
Using a Relational DBMS for CAD Data.
Technical Report UIUCDCD-R-90–1649, CS, U. of Illinois, Urbana, 1990.

[18] Xidak
Orion System Overview
Xidak, Inc., Palo Alto, CA, 1990.

[19] Zhang J.
Graphic User Display in Geographical Information Systems.
Master's thesis, Simon Faser University, 1991.

Appendix A Data Model

Block Object Data Model:

```
class Block {
    private:
        Vertex* block_center;
        char* name;
        . . . . . .
    public:
        char* Name();
        Bool On_Top_Of(Block*); // spatial predicate.
        Bool On_Bottom_Of(Block*);
        . . . . . .
        void Top_Of(Block*); // update function.
        void Bottom_Of(Block*);
        . . . . . .
        virtual void Translate(float, float, float); // Δx, Δy, Δz.
        virtual void Rotate(char, float); // coordinate and degree.
        virtual Face* Top(); // retrieve face.
        virtual Face* Bottom();
        virtual float Volumn();
        virtual char* Type();
}

class Cylinder : public Block {
    private:
        Face *top, *bottom, *side;
        . . . . . .
    public:
        void Translate(float, float, float);
        void Rotate(char, float);
        Face* Top();
        Face* Bottom();
        . . . . . .
        float Volumn();
        char* Type();
}

. . . . . .
```

Geometric Object Data Model:

```
class Geo_Obj {
    private:
        Vertex *center;
        . . . . . .
    public:
        virtual void Translate(float, float, float);
        virtual void Rotate(char, float);
        Vertex* Center();
        . . . . . .
}

class Face : public Geo_Obj {
    private:
        Vector* normal;
        char* color;
        . . . . . .
    public:
        virtual void Translate(float, float, float);
        virtual void Rotate(char, float);
        virtual void Boundary(Set&); // obtains the boudaries of this face.
        float Normal();
        char* Color();
        . . . . . .
```

```
}

class Curved_Face : public Face {
    private:
        . . . . . .
    public:
        virtual void Translate(float, float, float);
        virtual void Rotate(char, float);
        virtual void Boundary(Set&);
        . . . . . .
}

class Cylind_Surface : public Curved_Face{
    private:
        float Height(), Radius();
        Curve *top, *bottom;
        . . . . . .
    public:
        void Translate(float, float, float);
        void Rotate(char, float);
        void Boundary(Set&);
        . . . . . .
```

```
}

class Curve : public Geo_Obj {
    private:
        Vector* normal;
        float length;
        . . . . . .
    public:
        void Translate(float, float, float);
        void Rotate(char, float);
        Vector* Normal();
        float Length();
        . . . . . .
}
```

```
class Edge : public Curve{
    private:
        void LeftEnd(Vertex&); // obtains the left end of this edge.
        void RightEnd(&Vertex);
        . . . . . .
    public:
        . . . . . .
}
```

Appendix B Glossary

- AI Artificial Intelligence
- ASQL Application Specific Query Language
- CAD Computer Aided Design
- DB Database
- DBMS Database Management System
- DDL Data Definition Language
- DML Data Manipulation Language
- DSQL Dynamic Spatial Query Language
- GIS Geographical Information System
- GUI Graphical User Interface
- I/O Input/Output
- LN Less Normalized
- OID Object Identifier
- OIS Office Information System
- OO Object-Oriented
- OODB Object-Oriented Database
- OODBMS Object-Oriented Database Managament System
- OS Operating System
- PR Pure Relational
- RDBMS Relational Database Management System
- SQL Structured Query Language
- YACC Yet Another Compiler-Compiler