A64-2131

School of Computing
Department of Geographic Information Science

The Distribution of Geographic Information Systems Data in a Computer Communications Network

Bert Veenendaal

This thesis is presented as part of the requirements for the award of the Degree of Doctor of Philosophy of Curtin University of Technology

January 1999

Abstract

Geographic information systems (GIS) are developing in a rapidly expanding distributed environment. With the ever-increasing growth of computer networks and the Internet in particular, it is imperative that GIS take advantage of distributed data technologies to provide users and applications with shared and improved access to geographic data.

Geographic data distribution design is concerned with determining what data gets placed at which computer network sites and involves the issues of data partitioning, allocation and dynamic migration. Partitioning is concerned with how data, or fragments of the data, are apportioned to partitions. These partitions must then be assigned to network sites in an allocation process. Because data usage and access changes by applications in a dynamic environment, migration strategies are necessary to redistribute the data. In order for data distribution to reflect current usage patterns of applications, the design process must obtain and accumulate data usage information from applications.

This dissertation first details the predicate fragmentation (PF) model. The core of the model is the *PF tree* that has been developed and implemented to store and maintain usage information. User predicates, obtained from application queries, are inserted into the tree and *primitive predicates* can be identified from the tree. These primitive predicates define the fragmentation from which a data distribution can be determined. Predicate insertion and pruning operations are essential to the maintenance of the tree structure.

A methodology that uses the PF model to obtain a partitioning, allocation and migration strategy is then outlined. The fragments identified from the PF trees are aggregated into partitions that are then assigned to individual network sites using a *site access allocation* strategy. A dynamic migration strategy then uses changes in the PF trees to identify the data that must be migrated to a new site in order to accommodate the changing application environment.

The implementation of the geographic data distribution methodology is referred to as GEODDIS. The methodology was tested and evaluated using a mineral occurrence application called GEOMINE which was developed with the ArcInfo GIS. The results indicate that geographic data distribution performs well when successive applications have similar data usage requirements. For applications with very different data usage patterns, the performance decreases to the worst case scenario where all the data must be transferred to the site where it is used. The tree pruning and data migration operations are essential to maintaining a PF tree structure and distribution that reflects the current data usage of applications.

Acknowledgments

The completion of this dissertation would not be possible without the time and effort expended by the many persons involved. I would like to acknowledge the assistance of all those who provided information, ideas, constructive criticism and words of encouragement. In particular, I would like to thank Dr Doug Hudson and Dr Mark Gahegan for their advice, comments and suggestions. Their contribution was invaluable.

Furthermore, I would like to acknowledge the patience and perseverance shown by my family during these many years of "working on the thesis". Gratitude is expressed in particular to my wife Agatha and the children Janna, Alyssa, Jared and Sharene. Daddy will be spending much more time at home now!

Contents

List of Figures

List of Tables

1 Introduction

1.1 Geographic and Distributed Systems

As different technologies develop and progress, their integration must also ensue in order to consolidate the benefits of each. This is true for both *geographic information systems* (GIS) and *distributed data systems*. While both technologies have developed considerably in the past decade and much effort has been expended on their integration, further work is yet required in merging the technologies and capitalising on the benefits of their integration. Geographic data and systems must be taken into account in distributed computing environments, and distributed data systems technology must be embraced by developments in GIS. Distributed geographic information systems are not yet fully matured and many research issues and problems have yet to be addressed.

Managing data that is both geographic in nature and dispersed throughout a communications network presents many challenges to the database designer. Geographic data consists of spatial and attribute components and relationships that must be retained irrespective of where the data is placed in the network. GIS are required to handle large amounts of spatial and non-spatial data in a geographic database. Much time, effort and resources are necessary for managing and manipulating such data; database design is therefore of paramount importance.

Distributed systems and distributed databases are being developed as computer communications networks and open systems environments evolve. Database design methodologies must be adapted to accommodate both geographic and distributed environments. How should such geographic data be distributed within a distributed environment? In other words, how can distributed systems take advantage of geographic data and their properties and how can GIS databases take advantage of distributed systems developments? Such problems require a detailed examination of the issues, requirements, and solutions for geographic data distribution. An integrated approach to deal with the distribution of geographic data is referred to as *geographic data distribution design* and forms the focus of this dissertation.

1.2 Distributing Geographic Data

In most applications there is a relationship between the data and where it is used. This is due to different users and applications at various geographic locations accessing and querying different types of data in varying ways. To simplify access to the data by applications, the data ideally should be placed where it is being used. So in a computer network environment, the data would simply be placed at the site that uses it. However, data is usually not exclusively used by one user or one application. Rather, some data may be shared by multiple users and applications that are geographically dispersed throughout a communications network (Zwart and Coleman 1992; Coleman and Zwart 1992b; Bishr 1997). Determining the ideal location for the data may not be an easy process. A number of examples may make this clearer.

Consider a large forestry department with regional offices that are all linked onto one communications network. Each regional office manages the data within their respective regional boundaries. If the data within each region are not accessed by other regions then a data distribution strategy might be to allocate each region's data to the corresponding regional office. Each data feature in the database would be partitioned according to the regional boundaries and each data partition would be assigned to a different regional office (Figure 1.1). The partitioning used is a simple spatial partitioning based on location and usage.

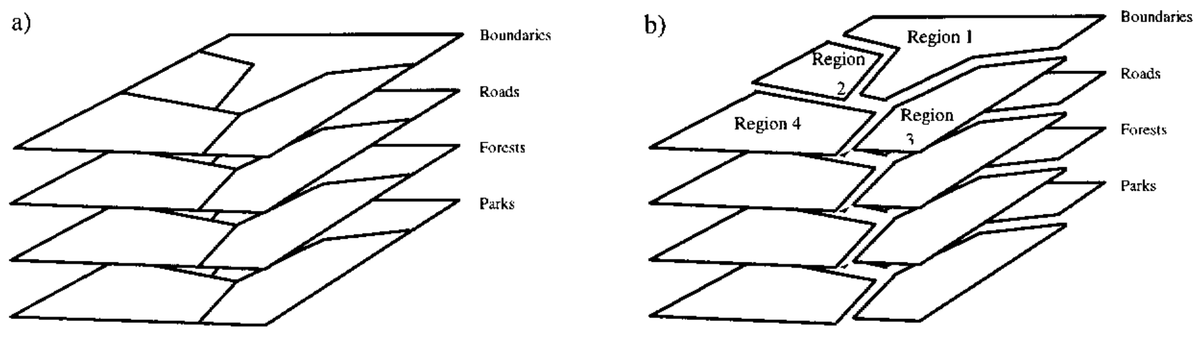


Figure 1.1 Partitioning by spatial boundary: a) before partitioning b) after partitioning

Now consider a second example of a planning organisation that consists of several departments, each with a computer connected onto the organisation's communications network. Assume that four departments exist, all dealing with different applications: roads and transportation, parks and reserves, urban industrial planning, and demographic analysis. If each application deals with a different set of data then a distribution strategy may be to allocate each dataset to the site within the department that uses it (Figure 1.2). The database is therefore partitioned according to feature type of the dataset.

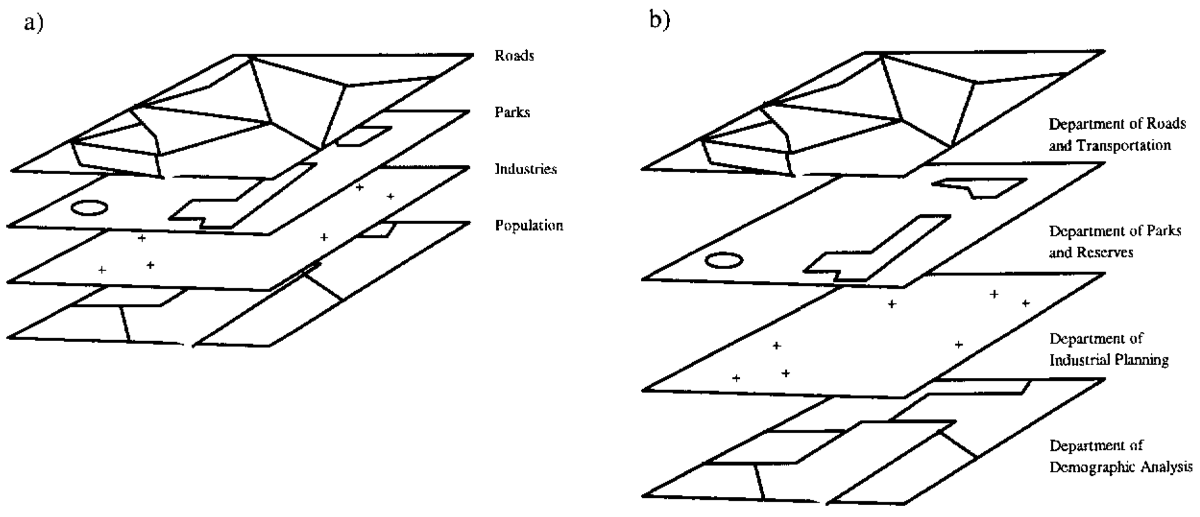

a)
Roads
Parks
Industries
Population
b)
Department of Roads and Transportation
Department of Parks and Reserves
Department of Industrial Planning
Department of Demographic Analysis

Figure 1.2 Partitioning by geographic dataset type: a) before partitioning b) after partitioning

In the latter example, data is partitioned according to a non-spatial attribute while in the first example spatial criteria were used. These examples represent the two extremes of data distribution and are generally not reflective of actual situations. In the forestry example, access to the data for a region is usually not limited to the corresponding regional office. Rather, access may also be required by other regional offices or by a head office that may require each region's data for summaries, reports, and planning. In the example concerning the planning organisation, there may be base datasets such as topography, cadastral boundaries, and administrative boundaries that are commonly used by all departments, in addition to the data that may be used exclusively by one or two departments. In such instances, a data distribution strategy is not intuitive and may be rather complex. Information regarding the usage of the data by applications and users could assist in determining a distribution strategy.

1.3 Overview of Geographic Data Distribution Design

Geographic data distribution design (GDDD) involves the structure and placement of geographic data within a distributed communications network environment. In other words, the problem is concerned with *what* data should be placed *where*. This, in turn, raises a number of issues involving the identification of the data to be distributed, determining the criteria for distribution, and making a decision for placement.

Often data distribution is performed in an *ad hoc* manner without any defined methodology. The data is placed on a local or most convenient site without any consideration for where the processing operations are located, where the data will be used, or what its frequency of access will be. In many cases, the only criterion used for the placement of data is the availability of storage space, often ignoring any consideration for future expansion. As the database grows in size and eventually exceeds existing storage limits then some or all of the database is redistributed, often again in an *ad hoc* manner.

A data distribution design methodology is necessary to identify and analyse the distribution requirements and produce an efficient distribution strategy (Figure1.3). Metadata involving application and database information must be used to determine the units of data to be distributed, and how these data units should be placed in a distributed network.

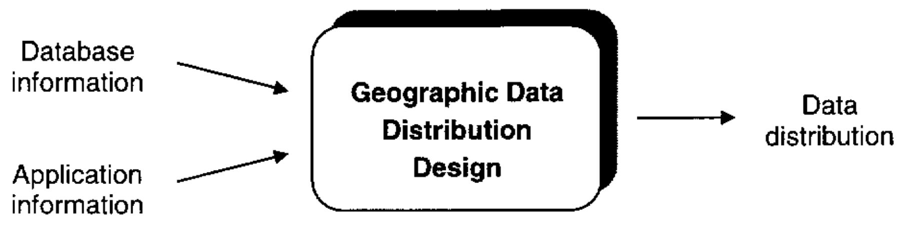


Figure 1.3 Geographic data distribution design

GDDD raises a number of issues. Firstly, the unit of data that is to be distributed must be determined. Should this unit of data be an entire database, a file, a complete entity or a fragment of an entity? Applications do not necessarily access an entire entity but often just a portion of it. It is therefore logical to expect that the distribution be based on fragments of the data related to their usage.

Different applications may access different, but possibly overlapping, data fragments. Thus a fragment will not necessarily have usage properties that are homogeneous within the fragment. Such a fragment must be further fragmented so that only those with homogeneous usage properties are identified. These properties can then be used to identify fragments that have similar usage properties and that can be aggregated into the same partition. Identifying the partitions of data is the second distribution design issue.

Horizontal and vertical partitioning are well-established techniques in database physical design (see Section 2.4.2). A partitioning should be based on the structure and usage of the data rather than be done in an arbitrary and *ad hoc* manner. For geographic data, entities can be partitioned according to spatial, attribute, or even a combination of spatial and attribute properties and relationships. The problem is to identify and incorporate these properties into the partitioning strategies.

Finally, the resulting partitions must be allocated to network sites. Data allocation should be based on site usage. A further consideration may be to allocate copies of the data to multiple sites. Some replication may already occur in the partitioning stage where common spatial primitive objects may be duplicated in multiple partitions (*eg*. endpoints of a line, common polygon boundaries). Further replication occurs in generating the multiple copies of data.

These issues have to be addressed as part of a data distribution design strategy. With geographic data, the problems increase in difficulty since the types of data and relationships, which must inevitably involve *spatial* properties, are more complex. Spatial data and spatial relationships must be considered when determining fragments, partitions, and allocations.

1.4 Significance of Data Distribution for GIS

GIS have developed rapidly over the past decade with a number of trends involving client-server and Internet technology leading towards distributed GIS. While traditional information systems handle textual data, GIS additionally incorporate spatial data along with attribute data and their inter-relationships. The manipulation, management and analysis of spatial data are much more complex than for textual data, involving greater volumes, more complex structures and additional operations. A distributed environment must be able to handle such spatial data.

Although storage space and processing speeds have improved dramatically, the demands of GIS have also increased since larger volumes of data, as well as enhanced and additional functionality, are sought by users and applications. The increasing storage and processing requirements of a GIS bring with it the need to share and optimise the use of computing resources. A distributed environment, providing the interconnection of geographically dispersed resources, allows access to resources previously inaccessible or under-utilised.

Data is a core component of GIS involving considerable effort and resources to capture and maintain it. It is not uncommon that well over fifty percent of time and resources required for a specific application involve building and maintaining a suitable geographic database (Devereaux *et al.* 1992). Much of the data maintained by GIS is multipurpose (*eg.* base data such as topography and thematic data such as soils and landuse) and can be used by many different users and applications. Data exchange and data sharing are therefore important aspects of GIS (Bishr 1997). Distributed network technology provides the potential for making data accessible and sharing it among those connected to the network. In addition, because such shared data can be made accessible from a non-local site, it need not be duplicated at every local site. A single copy or limited copies of the data can be accessed by all users irrespective of their location on the network, thereby improving the usage of storage resources and the management and maintenance of the data.

Considerable developments have occurred in the area of distributed systems and databases (Silberschatz *et al.* 1995; Adam *et al.* 1997; Waight 1997). Computer networking technology has evolved so that both local area and wide area networks are commonplace, enabling access to remote data and resources (see for example Newton *et al.* 1992, Alexandria 1996, and Adam and Gangopadhyay 1997). Distributed database management systems (DDBMS) are being developed to manage the distributed storage of data, access to data, and the transactions being performed on the data (Leigh and Burgess 1987; Ingres 1995; Oracle 1995a; Oracle 1995b; Smallworld 1994).

A significant development has been the World Wide Web (simply referred to as the *web*) taking advantage of the Internet interconnecting millions of computer sites across the world. The result has been an ever-increasing expansion of the Internet, servicing not only academic and governmental institutions, but also commercial and private interests. There are currently many tens of millions of users connected via more than 36 million host computers, and demand is expected to increase dramatically with over 100 million users before the year 2000 (Network Wizards 1998; Lewis 1996). By bringing the Internet and the web directly into households, the "distributed" community has been markedly broadened and exposed to a diversity of interests, requiring access to a vast array of data and information including geographic information (Vincent 1995). With such a development comes the need to manage and share data in an increasingly complex environment. More emphasis will be placed on the management of such data and systems as distributed technology continues to expand.

The developments in both GIS and distributed systems lead to further integration in these areas. Both technologies have much to offer and both can benefit from incorporating the other. Such developments include, for example, using a centralised data server to support several clients to handle spatial data processing (Zwart and Greener 1994), and extending existing GIS to support links to web-based applications and tools (see for example Dangermond 1996 and Roy *et al.* 1997). By making provision for spatial data and its properties, distributed systems are open to a wider array of applications for which geographic data is an important component. Using distributed technology within GIS can serve to improve the management and manipulation of what are often large volumes of geographic data that need to be shared among applications and users.

In considering the merger of GIS and distributed system technologies, geographic data distribution offers the following advantages (Ozsu and Valduriez 1991b):

> **Local Autonomy**. GIS databases are often managed and maintained by either the *data owner* or a designated *data custodian*. In a distributed environment, the data can be placed on sites where the owner or custodian has complete control. Individual organisations can therefore maintain autonomy as regards their data.
>
> **Accessibility**. Access to data is important, particularly for GIS users who require current data and for data commonly accessed by multiple users. By providing distributed access, the data can be placed on a site and shared by many users at different sites.
>
> **Expandability**. The increasing volumes of GIS data can be readily handled in a distributed system. The data can be stored at sites where there is available storage space, and additional storage and processing resources can be added to the computer network as required.
>
> **Improved performance**. Rather than deal with entire large datasets that often contain more information than is required, applications need only access the necessary subsets of data when they are required. A decrease in the amount of data being transferred results in an improvement in performance. Performance can further be enhanced since data distribution results in some parallelisation of processing; access to subsets of data at different sites can be executed concurrently.

Data distribution is not without disadvantages. There is a considerable amount of overhead involved in determining a data distribution design. Not only do the partitioning, replication, and allocation alternatives need to be considered, but also information for evaluating strategies must be obtained. Gathering such information may be difficult when predicting future data usage in a dynamic environment where applications and users are diverse and constantly changing. Yet such difficulties must be challenged and the obstacles hurdled as new strategies are researched and developed. The challenge of this research is to explore alternatives and determine strategies for GDDD (Veenendaal 1994c).

1.5 Research Objectives

The objectives of this research are as follows:

1. Identify the factors that affect geographic data distribution.

2. Find appropriate units of fragmentation for geographic data.

3. Produce a methodology for performing a geographic data distribution.

4. Evaluate the methodology by implementing and testing it.

1.6 Implementation

The methodology and solutions for a geographic data distribution strategy, as described in this dissertation, have been implemented in the GEODDIS (GEOgraphic Data DIStribution) system. GEODDIS is a suite of programs developed using the C programming language and ArcInfo Advanced Macro Language (AML) to test the methodology described in this dissertation. The programs have been tested using mineral occurrence data from a GIS application called GEOMINE and developed using the ArcInfo GIS software package.

Since it is impossible to exhaustively test the distribution strategies for every possible GIS dataset and application, a reasonable choice had to be made for the purposes of this research. ArcInfo is chosen as the GIS environment because it is one of the most popular and widely used commercial GIS, and because it incorporates a good transaction logging facility from which usage statistics can be obtained. The datasets and applications used in this research were chosen because of their availability, because they encompass a range of data and operation types and because they are typical of many applications. While it is difficult to define a "representative" GIS dataset and application, it is assumed that benefits of distribution design can be extended to many other GIS applications.

The GEOMINE project has been developed within the Department of Geographic Information Science at Curtin University of Technology and involves a mineral occurrence database and applications that operate on it. It provides a data integration tool and query interface for the database containing all the known mineral occurrences and mine sites in Western Australia (Veenendaal *et al.* 1995). The database presently consists of more than

8000 mineral occurrence locations along with relevant topographic information for the entire State stored in a geo-relational database within the ArcInfo GIS. A query interface tool has been developed for both ArcInfo, using the AML macro language, and ArcView, using the AVENUE programming language.

The entities of primary interest in the GEOMINE database are mineral occurrences referred to simply as *mines* in the entity-relationship diagram (ER diagram) shown in Figure 1.4. Note that the attributes of each entity include a spatial data type (*eg*. point, line, polygon or non-spatial), represented by a square box, and other attributes, represented by circles. Each mine point is related to a 1:250000 scale map sheet, of which there are 198 in Western Australia, a mine type (underground, bedrock or surface), status (developing or abandoned), and one or more minerals (primary, secondary or accessory). In addition, each mine point is related spatially by "nearness" to topographic features, such as roads. Other topographic features such as highways, towns, lakes, rivers, and national parks are included in the database but are not shown in the ER diagram. Additional features obtained for local areas of interest include native title regions, flora and fauna coverage and geology.

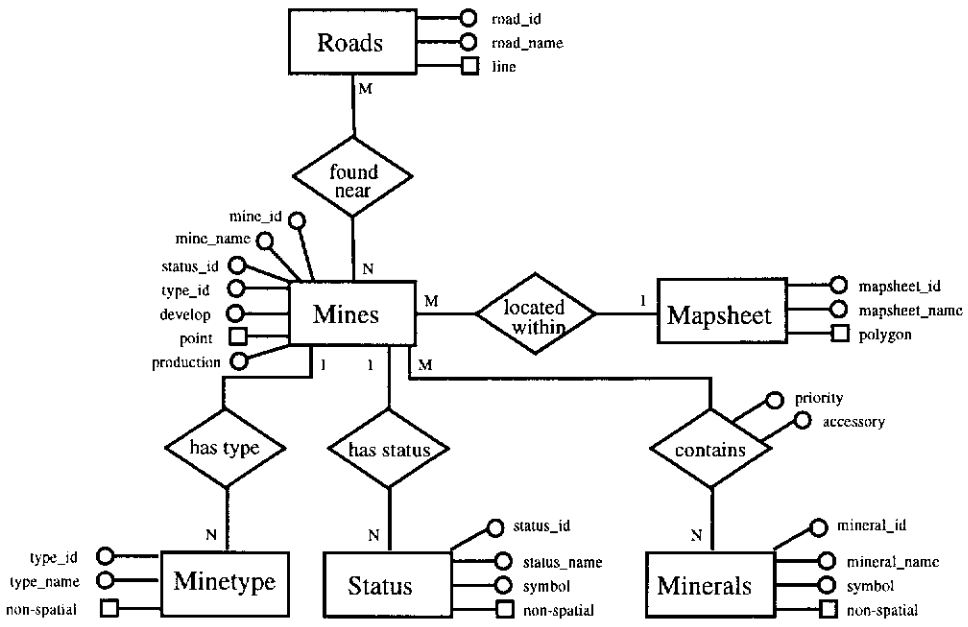


Figure 1.4 Part of GEOMINE data model

1.7 Thesis Structure

This dissertation is structured into six chapters. Chapter One introduces the problem of geographic data distribution and defines the goals of this research.

Chapter Two provides background information and reviews the literature regarding the data distribution problem. The distributed and geographic environments, the development of the problem and the issues affecting its solution are detailed.

A linked tree model, referred to as the PF tree, is described in Chapter Three and provides the data structure and operations to maintain the required information based on data usage and access patterns and used to determine a data distribution strategy.

Chapter Four details the methodology used to accomplish a geographic data distribution. The information obtained from the PF model is used in performing fragmentation and allocation of data to network computer sites. The testing methods are also discussed.

Chapter Five presents and analyses the results of geographic data distribution using the given methodology and provides conclusions.

Finally, a summary and further research directions are given in Chapter Six.

2 Background and Review of Data Distribution

2.1 Introduction

Many of the issues in data distribution have been in the research domain for over two decades. Early solutions involved only the allocation of files. As the problem further developed it became more complex, embracing issues such as data fragmentation, data allocation and dynamic data migration.

Even though the topic of geographic data distribution has not been extensively examined in the research literature, many related problems touch on the same or similar issues. These problems and solutions are considered in exploring data distribution and efficient access to geographic information.

The background of the data distribution problem and how it relates to geographic information are discussed in the following sections. The distributed data environment is first discussed with respect to its architecture, distribution design and the factors affecting and information required for data distribution design. Then the development of the distribution problem is explained before considering geographic data and the distribution issues related to both the attribute and spatial components. Predicates and selection operations, used to identify data fragments, are then described. Finally, the methodology for distribution design and a summary of the problem are synthesised.

2.2 Distributed Data Environment

The distribution of data is determined by various requirements and constraints that identify information about data, applications and network sites (Figure 2.1). Information such as data structure, usage patterns and location of applications indicate what data is to be distributed and where such data can be placed. Information regarding available storage, processing facilities, and how communication takes place between network sites may constrain the distribution, preventing data from being placed at particular sites because of lack of storage space or the high cost of transferring data.

Of course, the distribution strategies aim to provide an optimal data placement which achieves the distribution objectives. The objectives can include improving reliability and access to the data, reducing the costs of data access, and increasing performance by decreasing processing and response times. Importantly, to compare various design alternatives, there must also exist a means of determining the cost of data distribution based on the stated objectives (see Section 1.5).

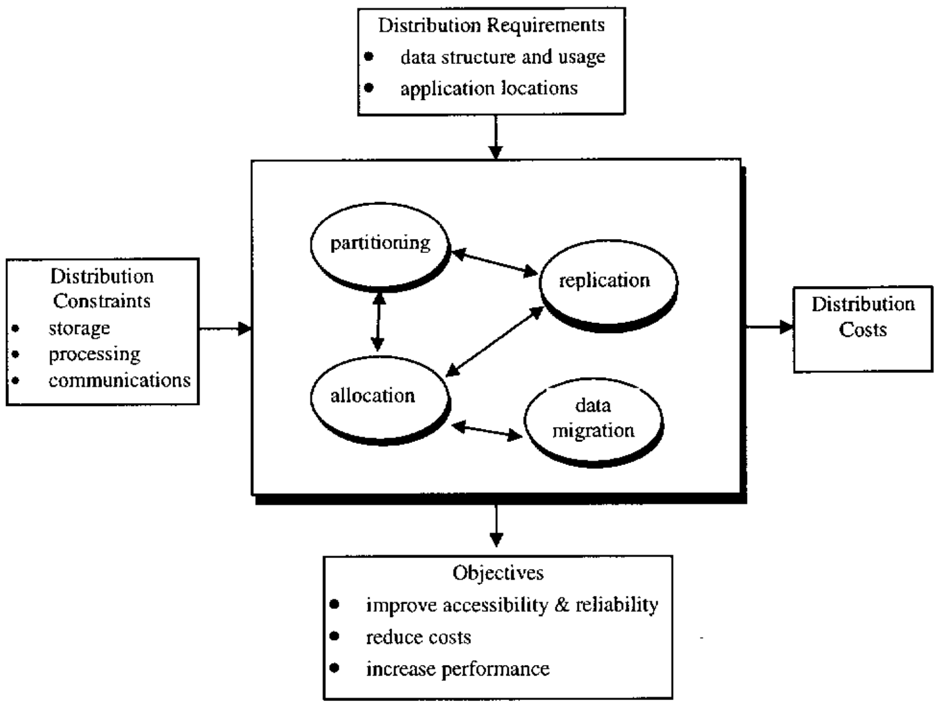


Figure 2.1 Aspects of data distribution design

2.2.1 Types of distributed data systems

Many types of distributed data systems exist involving database management systems that need not be the same at each site and with differing levels of autonomy. The various types of distributed data systems are distinguished by the manner in which data are accessed and managed. A system that allows data to be stored at more than one network location is generally referred to as a *distributed data system*. Location, functionality, and control are the three main factors that are used to identify distributed data systems (Wah 1981; Ceri and Pelagatti 1984; Ozsu and Valduriez 1991b; Bell and Grimson 1992).

Location refers to the network site where the data resides within a distributed system. The data is managed by one or more DBMS that can execute from one or more sites. The *functionality* provided by each DBMS at each network site indicates where the data may be processed. This information is used to determine which data must be transferred from the site where it is stored to the site where it will be processed (*ie.* a data *retrieval*) and which data must be transferred from the site where it is processed to the site where it is to be stored (*ie.* a data *update*).

Control of the data is an important factor as it indicates how data and access to the data are governed. Control can be either local or global. If control is *local*, each DBMS is autonomous and can keep track of its own data schemas and share the data with other DBMS when requested. If control is *global*, then the local DBMS has no autonomy but must cooperate in maintaining a global data schema as part of a global database.

Figure 2.2 illustrates a taxonomy of distributed data systems divided into homogeneous and heterogeneous types and further subdivisions based on the level of autonomy of a local site (Bell and Grimson 1992). *Homogeneous* systems are those where the same DBMS is present at each of multiple sites in the network, whereas *heterogenous* distributed data systems involve different DBMS at individual sites. Autonomy refers to the distribution of *control* and should not be confused with the distribution of *data*. *Distributed database systems* are a specific class of homogeneous non-autonomous systems that are tightly-coupled since the local DBMS are globally controlled (Kuspert and Rahm 1990; Hurson and Bright 1991; Orlowska *et al.* 1992). It should be noted here that the term *distributed database systems* is sometimes used ambiguously to refer to the general class of distributed data systems (Bell and Grimson 1992).

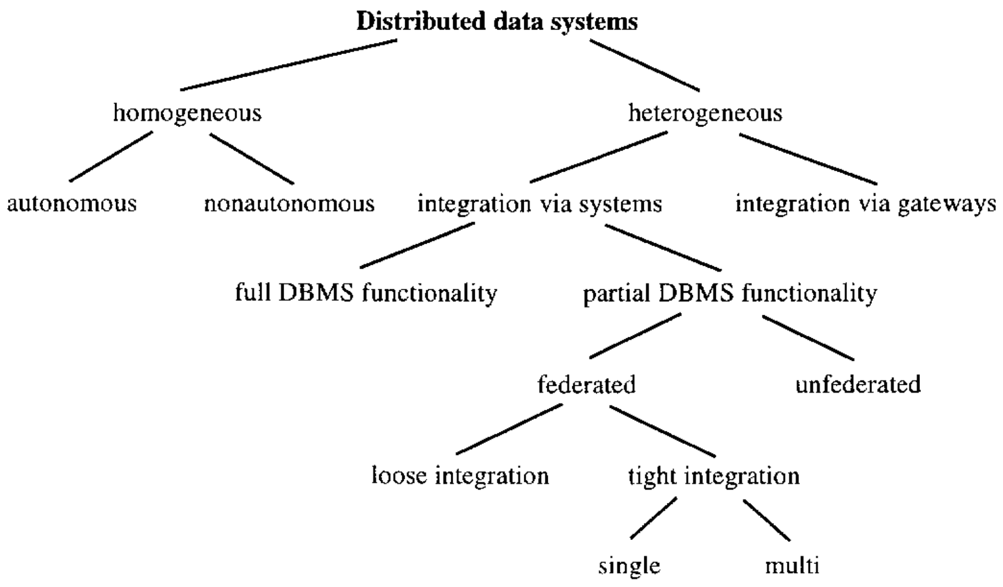


Figure 2.2 Taxonomy of distributed data systems (Bell and Grimson 1992, p. 45)

Heterogeneous systems integrated via gateways include distributed file systems (*eg*. NFS - SUN's Network File System) that share files across multiple networked hardware platforms. A *gateway* simply facilitates the transfer of a file from one platform to another. In such systems the local DBMS are completely autonomous and are not integrated at all since they *know* nothing about the existence of DBMS on other network sites. The method used to communicate data between DBMS is through data exchange. Often the DBMS incorporates some local tools to produce one or more files of data which can subsequently be transferred to a remote site and be imported into and interpreted by the remote DBMS (Thomas *et al.* 1990; Sondheim and Menes 1991; Satyanarayanan 1993).

Heterogeneous databases, where the DBMS functionality is partially integrated, are known as *multidatabase systems* (Hurson and Bright 1991) and can be classified into federated and unfederated systems. Local DBMS have no autonomy in *unfederated systems* while in *federated systems* the DBMS maintain their autonomy and yet participate in a federation to allow some global control of data sharing. Federated systems can be further classified to distinguish between *interoperable database systems* which use export schemas to provide a loosely-integrated federation, and *tightly-coupled federated systems* which allow the creation of single or multiple federated schemas (Sheth and Larson 1990).

The type of distributed data system influences the DBMS functionality available at each site and, hence, has an impact on where data can be transferred and stored. A more tightly integrated system with less local autonomy allows for better global control of the data distributed across a computer network. Such global control includes the aspect of deciding where the data is to be placed. For autonomous DBMS, the decision for data distribution is made locally and must involve interaction between the local DBMS to come to a mutual agreement. Stonebraker *et al.* (1996) propose to use an economic paradigm for the Mariposa distributed DBMS where autonomous sites negotiate by *buying* and *selling* data and *bidding* on queries which access the data.

2.2.2 Distributed architecture

Data modelling is the process of determining which entities and relationships are important and how they should be structured. The user or application views the data at a higher level of abstraction than the manner in which it is physically stored in a database. For example, an entity viewed externally by an end-user may be a road or mineral deposit. Internally, within the database, the data may appear quite differently - the road may be represented as a sequence of line segments and the mineral deposit as a single point.

Within relational DBMS, data modelling is often based on the ANSI-SPARC three-level approach as depicted in Figure 2.3 (Date 1982). The external level defines the model as the users and applications observe it. This is the view that most closely matches a specific user or application. At the conceptual level, a common or global view is defined on the data that incorporates all the data models as defined for the various external views. The internal view defines how the conceptual view is mapped onto a physical representation in a computer.

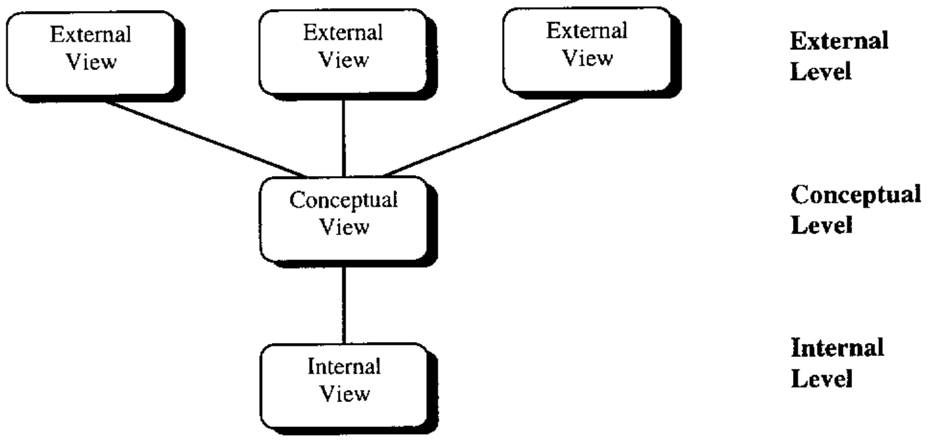


Figure 2.3 ANSI-SPARC data model framework

With distributed systems the architecture must be modified to incorporate the internal views at each local site. Many architectures have been proposed for distributed systems (Ceri and Pernici 1985; Ozsu and Valduriez 1991b). Ceri and Pernici (1985) present a model where a new level known as the *logical level* is placed between the conceptual and internal levels (Figure 2.4.). This level defines how the conceptual data model is mapped to each site in a distributed data system. At the internal level, this logical model is mapped onto a physical model at each computer network site.

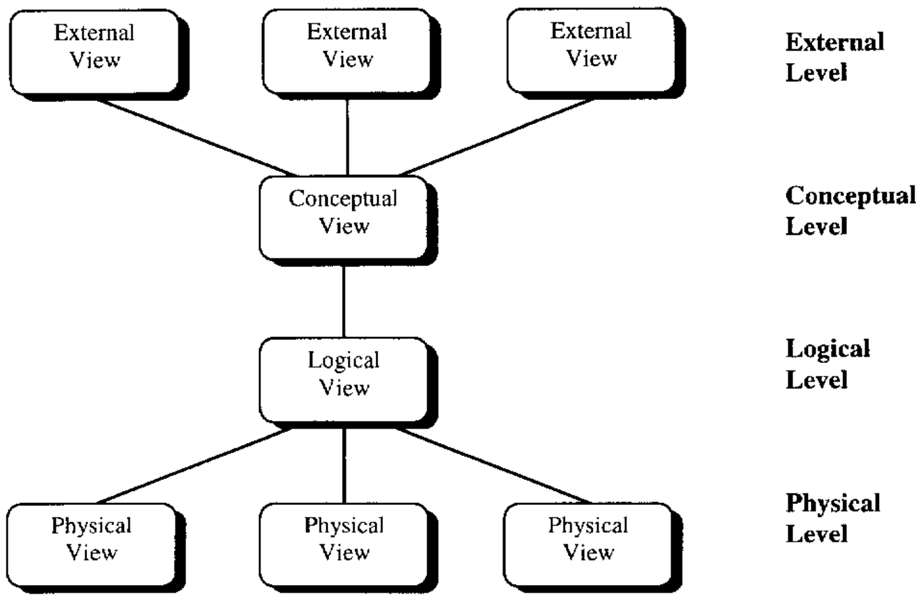


Figure 2.4 Distributed architecture with a Logical Level

An alternative architecture provides for different conceptual views at the local level, each of which is mapped directly to the local physical view (Figure 2.5). A *global* conceptual view is derived from these local conceptual views. The external view can be either global or local.

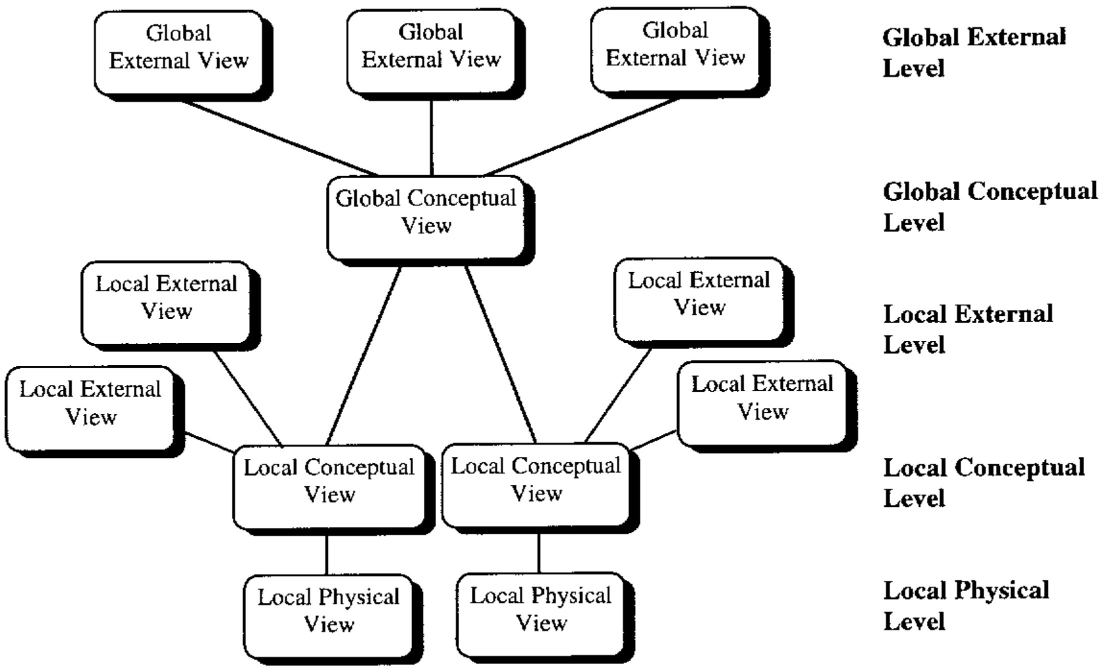


Figure 2.5 Distributed architecture

The effect of a data distribution is that a physical view is provided at each individual network site. The use of the data by users and applications at the external level must therefore translate to a distribution at the logical level that is implemented in multiple views, one at each site, at the physical level.

2.2.3 Objectives of data distribution design

The objective of data distribution design is to determine an optimal allocation of data to computer network sites. An optimal distribution can be measured in a number of ways, namely: improved performance, reduced costs, and increased availability. While a distribution strategy that achieves all three objectives is ideal, it is not always possible since these objectives are sometimes in conflict. A balanced solution will then have to be found.

Performance

Two of the common performance objectives are minimising response time and maximising system throughput (Dowdy and Foster 1982). For data distribution, the response time may be improved by reducing the time required to access the data. Ideally, all the data should reside at the site where it is being processed. However this is difficult if not impossible to achieve when the same data is being used by multiple applications on different sites. Rather, the data should be stored as close as possible to where it is being used (Ozsu and Valduriez 1991a). The "closest" that data can be stored is directly at the site where it is used. The next "closest" refers to a site from which the data can be accessed with minimal time.

The **locality of reference** or simply **locality** of the data refers to the site(s) where the data is frequently accessed by user applications over a given period of time (Ceri and Pernici 1985; Copeland *et al.* 1988; Gavish and Liu Sheng 1990). Ceri and Pelagatti (1984) refer to this as **processing locality**. If only part of the data is accessed, then the frequency is multiplied by the fraction of data being accessed to determine the locality. A measure that can be used to determine the locality is **polarisation** which indicates the deviation from a uniformity assumption (Ceri and Pernici 1985). Polarisation is measured as the percentage of data accesses that are addressed to a data fragment at a particular site. Data can be distributed based on its **maximal locality** which indicates that the data is assigned to the site where it is most often processed (Ceri and Pernici 1985). The result is that fewer remote data references are required leading to faster data access and improved performance.

By distributing data fragments over multiple sites, the workload in accessing and processing the data can be shared. Data fragments required by an application and stored at different sites may be accessed concurrently. Further, it may be possible to process these fragments in parallel at multiple sites by one or more applications, thereby improving the level of concurrency (Wong and Katz 1983; Ceri and Pelegatti 1984; Ozsu and Valduriez 1991b). The result is that system throughput can be increased.

Costs

Another way of optimising data distribution is by minimising costs. Data distribution involves a number of costs: communication, processing, concurrency control and integrity constraint checking. Communication costs are incurred for data fragments that must be retrieved from remote sites and processed by an application on a local site. Similarly, any updates that must be propagated back to the remote sites where the data are stored will incur communication costs. These costs depend on the amount of data being transmitted and the network characteristics (Ceri and Pernici 1985).

Multiple data fragments that have been retrieved from several sites may need to be merged before being processed at a local site. For an update operation the data may have to be split up into fragments before being transferred back to the remote sites on which they reside. CPU and I/O processing costs are involved in this process of merging and splitting of fragments.

Concurrency control involves synchronising access to the database while maintaining its integrity (Ceri and Pelagatti 1984; Ozsu and Valduriez 1991b). For a distributed database system this includes managing access to multiple copies of the data. Integrity constraint checking involves maintaining consistent database states where information on data relationships and semantics are used to ensure data correctness. Integrity constraint checking is costly and even more so in a distributed system where data fragments to be checked must be accessed from multiple sites.

Availability

An objective of distribution design may be to improve availability and reliability. This can be achieved by introducing replication of data fragments across multiple sites. Thus, applications requesting data can access alternate copies if a specific copy is unavailable due to some network or other failure. Additionally, if a copy of the data becomes corrupted due to some hardware or software failure, alternative copies of the data can be accessed and used to maintain data reliability.

Ideally, all these objectives should be aimed for in a geographic distribution design strategy since they are all important. A design strategy should result in reduced costs, improved performance, and increased availability. They are also often interrelated. For example, by placing data on a site according to its maximal data locality, the communication costs may be reduced resulting in faster response times and hence improved performance. Also, by replicating data on an additional site, multiple applications can access the data concurrently, improving system throughput and also ensuring data are available if a failure occurs or data becomes inaccessible at one of these sites (Chu and Hurley 1982).

These objectives can also sometimes conflict with each other. For example, by replicating data to improve data availability, further communications costs may be incurred due to a high occurrence of updates. As another example, data may be further fragmented and allocated to as many sites as possible to achieve a high level of concurrent data accesses and a faster response time. However, the costs involved in merging and splitting the fragments for processing may be substantially increased.

To manage the conflicts, all these objectives must be balanced with respect to one another. Often, during the design stages of an information system, cost measures are used to evaluate the system and performance is a secondary issue. Once the system is operational, more information is known about the workload so that more emphasis can be placed on performance (Dowdy and Foster 1982). All these objectives should of course be included in an optimisation model for data distribution design. However, in practice this is difficult to achieve due to the complexity of integrating them (Ozsu and Valduriez 1991b). Such a model essentially would involve integrating the partitioning, allocation, and replication problems rather than treating them as separate issues to be tackled consecutively.

2.2.4 Factors Affecting Data Distribution

A data distribution design is influenced by many factors involving organisations, users and applications. Factors may include organisational structures and policies, inter- and intra-organisational data flows, the GIS project implementation stages (*eg*. data capture, query, analysis), GIS application types (*eg*. project-based or inventory-based) and types of users (*eg*. developer, end-user, professional, manager, planner). These factors are identified in Figure 2.6 as *high-level* factors relating to external elements which contribute to data usage.

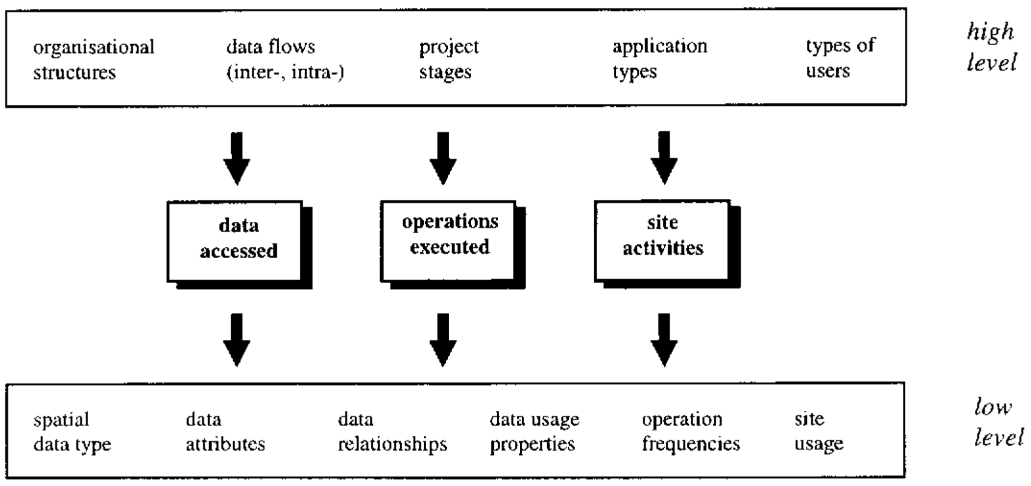


Figure 2.6 Various levels of factors affecting data distribution

For example, departments within the same organisation but involved in different applications may access a common database from different computer network sites in the organisation. Projects at a data capture stage will require different access to those focussed on spatial analysis. Managers requiring summary and statistical information use very different applications than do database developers involved in editing data and verifying data relationships. Data flows between organisations may lead to specific data entities being accessed more frequently than other entities. Organisational policies may place constraints on how a database can be distributed by dictating where certain databases must be placed or maintained (*eg.* within a particular branch or regional office).

The high-level factors affecting data distribution determine the "*what, how and where*" of data access. It is important to determine *what* data is being accessed since it is this data that will have to be fragmented and allocated to network sites. *How* the data is accessed by operations from users and applications will determine how the fragmentation is to take place and *where* the data is accessed is necessary to decide at which site(s) the fragments are to be allocated.

Figure 2.6 also shows some *low-level* factors that identify usage information regarding data, applications and sites. These factors specify information that may be too difficult to determine directly from the high level factors. As an example, suppose that information on a collection of mines is required by a mining organisation. The data, operations and sites involved depend on a number of high-level factors including: who requires access, for what stage of the project the data was used and which department has control of the data. It would be very difficult (if not impossible) to use the high-level factors directly in determining a data distribution. Rather, low-level factors such as the frequency of access of mines at each site, the operations executed on the mines and the computer site(s) at which the mine data are located, can be used directly by the data distribution process.

Ozsu and Valduriez (1991b) correctly point out that too many factors contribute to an optimal design. Goodchild and Rizzo (1986) indicate the difficulty with obtaining high-level information concerning profiles of GIS users and applications. Obtaining and modelling such information and relating it to data usage patterns over time could be the subject of further research. Identifying all the high-level factors and using them to determine a distribution strategy is virtually impossible. Instead, *low-level* factors, which are much easier to identify, should be used in the data distribution process (Veenendaal 1994a; Coleman *et. al.* 1992). Information contained in the high-level factors is implicit in the low-level factors. This research uses information at the low-level to decide on a data distribution. The low-level factors are detailed in Section 2.4.6.

2.2.5 Data and query distribution

The location of data among the sites of a distributed data system depends on the locations at which the data is required for queries generated by users and applications. However, the queries themselves can also be distributed so that the location at which processing occurs depends on the location of the data being accessed. Data and queries are so closely related that the distribution of one affects the distribution of the other (Apers 1982; Ceri and Pernici 1985; Goyal *et al.* 1993; Wilschut 1994). However, finding an optimal solution in simultaneously distributing both data and queries is a difficult problem.

Obviously the placement of data and the location at which processing occurs will influence the cost of data access. If the required data and the queries being executed are both located on the same local site where the corresponding user application resides, then we assume here that there is no cost involved since there is no non-local data being accessed. However, when data and/or queries are located at one or more sites other than the local site, then costs relating to the transfer of data between sites are incurred. Figure 2.7 depicts various scenarios involving different locations for data and queries.

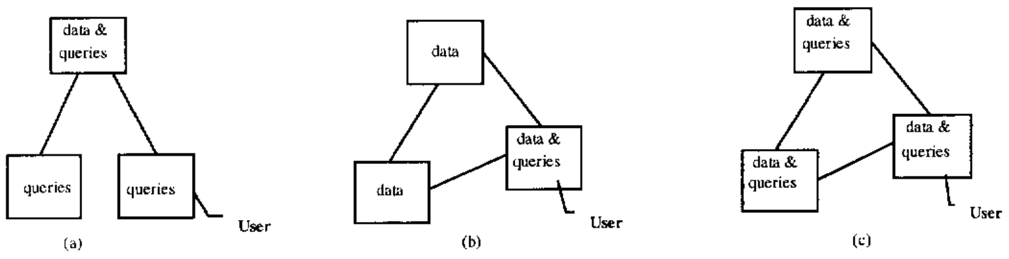


Figure 2.7 Scenarios for data and query distribution

Figure 2.7a represents the typical client/server architecture where the data resides at a central server. When a user application located at one of the client sites makes a request, only the required data is extracted from the database by server operations and retrieved from the server. The result is then transferred to the client where it is used and updates, if any, are returned to the server. Such an architecture was trialed by Zwart and Greener (1994) for GIS data over a wide-area network and is being implemented in GIS products (Waight 1997). Because all the data is centrally stored at one location, data distribution is not an issue.

Another scenario involves data that is not centrally stored. Such an environment may involve *local query execution* where the data is found at multiple locations and the query gets executed only at the local site where the user has initiated it (Figure 2.7b). This is similar to many current networked workstation environments where users execute the queries at their own workstations and access data found at various network locations through distributed file system services such as SUN NFS (Sandberg *et al.* 1985; Bishr 1997). In such a case, all the requested data must first be transferred to the local user site before the query gets executed, and any updates, if required, are transferred back to the appropriate sites. With distributed file systems, the data are actually entire files. Other than for file retrieval and update, no further processing is required at the remote sites. Such systems are contained in the class of loosely-coupled, heterogeneous distributed data systems (Section 2.2.1).

Each site can also function as a client/server as illustrated in Figure 2.7c so that both data and queries can be found on any site in the network. The question now concerns what queries can be executed at each site and who controls where the queries are executed; in other words, what database management system functions occur at each site and how their execution is controlled. In the simplest case, when an application is initiated at the local site, the required subset of data is retrieved from each of the remote sites at which it resides and is transferred to the user site where it is executed. Any updates and changes must be transferred back to the appropriate sites where the database is maintained.

A more complex situation is where the application queries execute on multiple sites and transfer the results back to the local user site (see, for example, the *Virtual Database* proposed by Abel *et al.* 1994). This involves *query processing* where a query execution plan must be constructed based on the location of the data and the nature of the query (Chu and Hurley 1982; Kriegel *et al.* 1991). A change in location of the data may result in a different query execution strategy. The control of the query processing may be global or local, depending on the autonomy of the DBMS at each site. These systems range from homogeneous distributed database systems to heterogeneous multidatabase and federated systems.

Formulating a problem to combine data and query distribution, and obtaining an optimal solution is difficult. The problem can be simplified into two problems: the data distribution problem where the location of the queries are known, and the query distribution problem where the data locations are known (Apers 1982; Ceri and Pernici 1985; Goyal et al 1993; Ozsu and Valduriez 1991b; Wilschut 1994). In this dissertation we deal with the data distribution problem and assume that the query locations are known or can be determined.

One further aspect that should be noted is that data distribution and query distribution often occur at very different time intervals. For each query being executed, the query distribution must occur before the query can be completed. In contrast, data distribution need not be done for every query and would normally occur at much less frequent intervals. Data migration is expensive, particularly for large databases, and therefore would occur only after many queries, or even applications, have been executed.

2.3 Development of the Distribution Problem

The data distribution problem has developed in relation to the unit of data to be distributed. This unit can be a file, a database entity or a fragment of data. In this section the development of the problem is reviewed.

2.3.1 File allocation problem

The original data distribution problem is known as the *file allocation problem* (FAP) and involves the allocation of complete files to computer network sites. The FAP was first identified by Chu (1969) who used mathematical programming techniques to define a solution. The criteria for determining an allocation included file size, storage capacity on each site, and data transmission costs. The unit of data being allocated is the entire file irrespective of its contents and the organisation of data that is contained within.

One early simplified model was developed by Casey (1972). The objective was to minimise the cost of satisfying all queries from all nodes and the cost of storing and updating the file. File *retrievals* were distinguished from file *updates*. Because file replication was allowed, the number of copies and their locations also had to be determined.

Laning and Leonard (1983) extended this approach by allowing multiple files with the constraints of a fixed number of file copies and equivalent location costs for each site. However, not all files need to be considered since Laning and Leonard postulated that usually about 20% of the files account for 80% of the storage, and of these 20%, many have obvious locations due to security or usage reasons. Therefore only relatively few files need be included in the file allocation problem. More recently, others have extended the FAP for multiple copies so that consistency is maintained among the replicated data (Tewari and Adam 1992).

Morgan and Levin (1977) extended the FAP to include the dependencies between files and programs. The file storage, program storage, and communications costs are considered as well as the file retrieval and update costs. The retrievals and updates originate from the *request* site, are executed at the *program* site, and access the data at the *file* site (Figure 2.8). So for a retrieval, the file must be transferred from the *file* site to the *program* site. If an update occurs, then the file must be transferred back to the *file* site. If copies of files are found at multiple sites, then the update cost must include the transfer of the file to each of these sites. Using this model, the placement of the data can be related to where the query is executed. Once again the unit of transfer is the file, irrespective of the contents and organisation of the data contained. If even only a very small portion of a file is being queried or updated, the entire file must be transferred to the *program* site and then back again to the *file* site for the update.

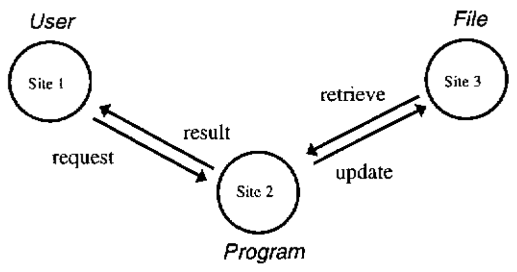


Figure 2.8 Dependency between files and programs

Further heuristic algorithms to solve the problem as identified by Morgan and Levin have been developed (Fisher and Hochbaum 1980). In addition to the dependency between files and programs, Yu *et al.* (1984; 1985) considers the dependence between files due to retrieval and update queries, although they limit their work to star topology computer networks involving a central node and regional sites. Gavish (1987) considers the locations of data sources, of end users, and of reports generated for end users and that require access to the database. This information is used to determine the locations of databases and how data sources are to be assigned to those sites.

The file allocation problem has been proven to be an NP-complete problem (Eswaran 1974). NP-complete problems have the characteristic that as the size of the problem increases linearly, the complexity of the optimal solution increases exponentially (Hevner and Rao 1988; Sedgewick 1988). Some of the classical problems such as the warehouse location problem and the knapsack problem are NP-complete (Wah 1981; Ramamoorthy and Wah 1983; Ozsu and Valduriez 1991b). Because of the difficulty in obtaining an optimal solution, efficient heuristic approaches have to be used to obtain reasonable solutions (Fisher and Hochbaum 1980). The literature provides some good surveys of the file allocation problem and heuristic solutions (Wah 1981; Wah 1984; Dowdy and Foster 1982; Hevner and Rao 1988; Boffey 1992).

An extension to the file allocation problem is the re-allocation of data as the file access environment changes over time. This is referred to as *dynamic* file allocation, in contrast to *static* file allocation, and involves using information about changes in data access rates to determine a new file location (Morgan and Levin 1977; Yu *et al.* 1985). Gavish and Liu Sheng (1990) distinguish between *dynamic file allocation* and *file migration*. The former refers to a system-wide file re-allocation that applies to problems with long time horizons of months or years and incurs a high cost. The latter refers to the re-allocation of a single file copy that incurs relatively little system interruption and can be performed frequently. These dynamic allocation problems, similar to their static counterpart, are NP-hard in their complexity. Good reviews of the dynamic FAP and file migration problems are provided in Hevner and Rao (1988) and Gavish and Liu Sheng (1990).

2.3.2 Database allocation problem

In a distributed environment, data access cannot simply be equated to remote file access; rather, access is based on the data entities and their interrelationships. Thus the allocation problem, referred to as the *database allocation problem* (DAP), is based on a logical data unit such as an *entity set* rather than being based on a physical data unit such as a file. The solutions for the FAP can, in general, be applied to the DAP (Hevner and Rao 1988).

An *entity* is a real-world phenomenon of interest that can be represented and stored in a database (Chen 1976; Laurini and Thompson 1992; NCGIA 1990). In this dissertation the word *object* is referred to as the representation of that phenomenon in the database. Such an object is not to be confused with an *object* in the "object-oriented" sense of the word. Entities can be classified into *entity sets* that can be represented as a *table* or *relation* in a database. A specific entry in an entity set is referred to as an *entity instance*. For the sake of simplicity and where the meaning is obvious or unambiguous, the term "entity" will be used in this dissertation to refer to "entity set".

Chang and Liu (1982) recognise that the allocation should be performed on a relation rather than simply a file. Relations are allocated to sites based on *access probabilities* that are obtained from usage statistics associated with a user's view. A fully connected network with identical transmission costs between sites is assumed. Transmission cost is therefore a function of the amount of data being transmitted. Both retrieval and update costs are considered. Constraints such as storage limitations at each site and a maximum allowable limit on the number of data copies are imposed. The problem is transferred to a network flow problem where relations and sites, represented as network nodes, are fully interconnected. The cost of allocating the relation to a site, based on usage, is associated with the network edges. The cost of the flows from relations to sites is then minimised to obtain a solution.

The database allocation problem is considered by Gavish and Pirkul (1986) with regards to applications involving banks with branches dispersed over a geographic area. Transactions by customers exhibit a strong locality of reference since most occur in the vicinity of the branch where the customer's account is located. The database is therefore partitioned according to bank branches. Essentially, the database is partitioned into smaller databases each representing all the customer accounts at a specific branch. These databases are then assigned to networked computer sites located within the branches.

These DAP solutions fragment a database to the level of an entity. Chang and Liu (1982) in dealing with the allocation of relations, recognise that databases can be vertically or horizontally fragmented. They use the term *database decomposition* which was first introduced by Chang (1972). Decomposition, more commonly referred to as partitioning, extends the DAP problem by introducing fragmentation of entities.

2.3.3 Data distribution problem

Rather than requiring access to an entire entity, application programs often access only a subset of the entity. This smaller subset could be a group of records (*eg.* rows in a relation), a group of attributes (*eg.* columns in a relation) or some combination of entity instances and attributes. Therefore, when distributing the data, it is logical to deal with fragments of entity sets that more closely correspond to the manner in which data are accessed and used. This is the basis of the *data distribution problem* (DDP).

Unlike the FAP and DAP, which deal with fixed units of distribution (file and entity respectively), the DDP must consider strategies to fragment a database and determine the unit of distribution. Solutions to the DDP are therefore more complex and involve a number of issues (see Figure 2.9) that must be properly handled in the design of a solution (Ceri and Pelagatti 1984; Ceri *et al.* 1987; Hevner and Rao 1988; Veenendaal 1990). These issues are explained in Section 2.4.

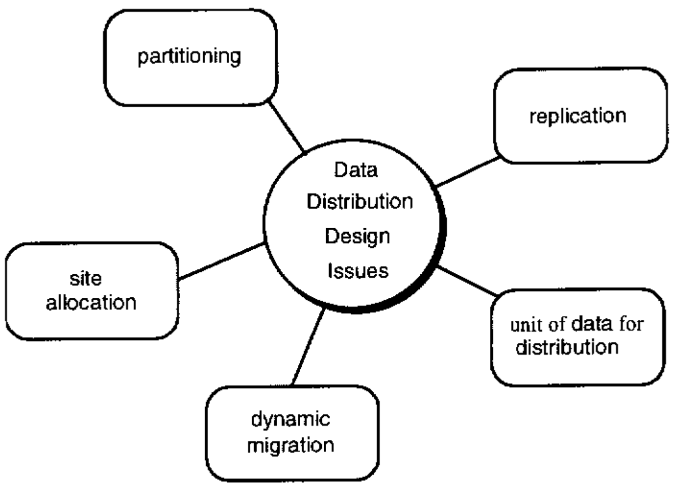

partitioning
replication
Data Distribution Design Issues
site allocation
unit of data for distribution
dynamic migration

Figure 2.9 Data distribution design issues

A substantial amount of research has been expended into developing solutions for the DDP (Navathe *et al.* 1984; Sacca and Wiederhold 1985; Ceri and Pernici 1985; Cornell and Yu 1990; Ozsu and Valduriez 1991b; Wolfson and Jajodia 1992). Further information and reviews of research into data distribution design can be found in Wah (1981), Dowdy and Foster (1982), Hevner and Rao (1988) and Ozsu and Valduriez (1991b).

2.4 Overview of Geographic Data Distribution

The first issue to be addressed in data distribution design is deciding on the unit of data to be distributed. The unit of data is termed the *cell* and involves the largest subset of data that is handled as a unit by any user application (Hua *et al.* 1993). A cell can be a file, a relational table, an individual entity, or a fragment of an entity that can eventually be assigned to a *partition* of data on a network site. A *fragment* refers to a subset of an entity (or relation) and the process of dividing an entity into multiple subsets is termed *fragmentation*. Figure 2.10 illustrates the relationship of data entities to cells, partitions, and sites. A cell can contain one or more entities or fragments of entities.

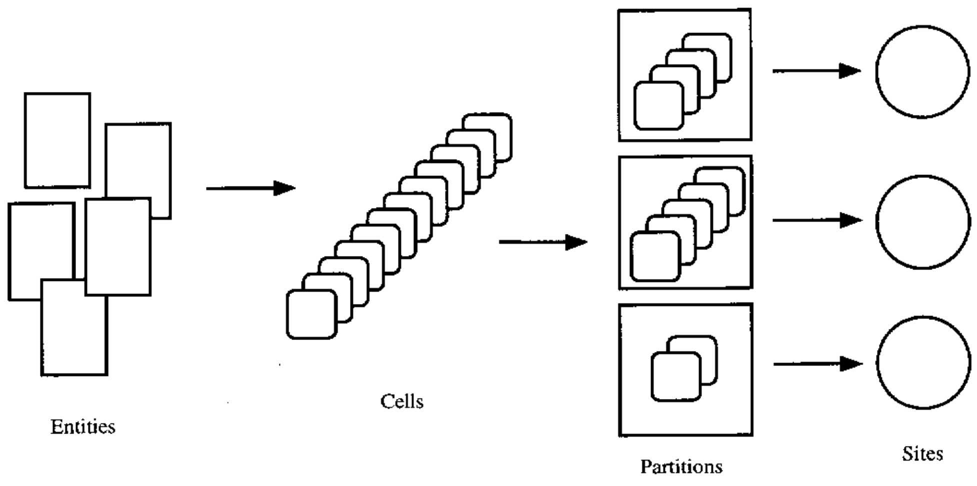


Figure 2.10 Partitioning and allocation

The whole process of building partitions from entities is known as *partitioning*. The end result is a subset of data referred to as a *partition*. A partition can consist of one or more cells aggregated together. The constructed partitions can then be assigned to a site. This latter step is known as data *allocation* or *placement*.

A further issue involves the possible *replication* of data among the sites of a network. Providing copies of the data on multiple network sites is sometimes desirable to improve data access from different locations within the network. Improved reliability is also a result since, if the data are unavailable from one site due to a site failure, another copy is available at another site. Some data replication is inevitable in the distribution process, as is the case with vertical fragmentation (see Section 3.3.2).

The final site allocation of partitions is based on how the entities are accessed. For many types of applications, data usage will change over time as different applications and users access the data from various network sites. Thus, this "final" allocation will have to be modified from time to time using *data migration* strategies to incorporate changes in usage patterns.

These distribution problems are further detailed in the following sections. This dissertation addresses the fragmentation, allocation and data migration problems in a methodology for geographic data distribution. Although they are referred to, data replication and its associated issues are not addressed here in full but require further research to effectively incorporate them into the methodology.

2.4.1 Geographic data

Data that has some spatial property is referred to as *geographic data*. Geographic data may consist of both spatial and non-spatial data. Often geographic data are ambiguously referred to as *spatial data*. However, there is a distinction between spatial and non-spatial data, and both are used to represent a geographic entity. Therefore, throughout this dissertation, the distinction is maintained by referring to the spatial component as *spatial data* and the non-spatial component as *attribute data* (Figure 2.11). The term *geographic data* will encompass both aspects (Veenendaal 1997).

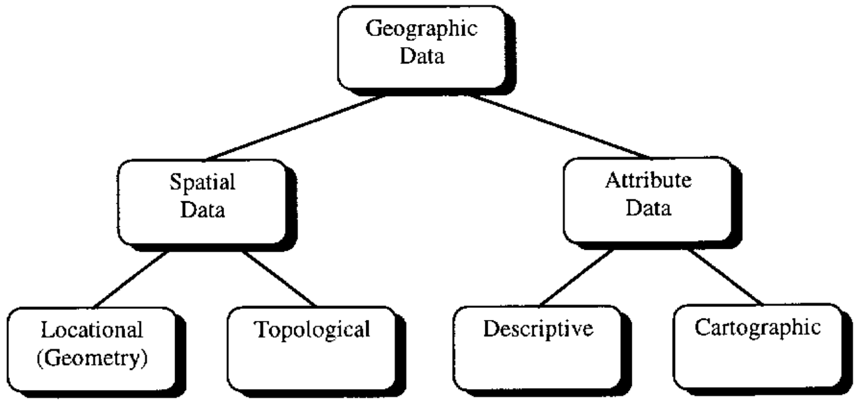


Figure 2.11 Geographic data components

Spatial data refers to the spatial component of the data with properties such as location and/or topology. *Location* refers to a geographic location usually in two or more dimensions, thereby defining the *geometry* (position and shape) of a spatial data entity. A spatial data entity is represented using spatial data primitives such as points, lines, polygons or raster cells. *Topology* refers to the relationships between spatial data primitives and is based on properties such as connectivity (adjacency and incidence), enclosure and orientation (Laurini and Thompson 1992; Adam and Gangopadhyay 1997). Two types of data structures are used primarily for representing spatial data: vector and raster.

For the vector data structure the basic primitives used for representing entities are points, lines, and polygons. In essence the spatial primitive is the *line*, with a point being a 0-dimensional line and a polygon being a closed set of adjacent lines. Within a layered architecture, different *geographic entities*, also referred to as *geographic objects* or *features*, are represented in different layers. The spatial data component of vector data is closely linked to any associated attributes of the entities. In the geo-relational model, the spatial database is linked to a relational database containing the attributes. Because of the close link required in spatial analysis, some GIS store the spatial data with the attribute data in an extended relational database. Some vector GIS explicitly store topology (*eg*. ArcInfo) while others simply calculate topology when required (*eg*. MapInfo).

The raster structure consists of a regular tessellation of geographic space for which values are defined to represent the presence or absence of an entity or its attribute value, or some continuous thematic value. The basic unit of data on which the tessellation is based is the *grid cell*. Different types of entities and different attributes for entities are usually represented in different layers. Both the spatial and attribute components of raster data are linked to the tessellated structure.

A good overview of vector and tessellated data structures is provided by Peuquet (1984) and further detail may be obtained from Burrough (1986), Laurini and Thompson (1992), Worboys (1995) and DeMers (1997). This dissertation is concerned with the topological vector structure where both entities and their relationships must be maintained in a data distribution strategy.

2.4.2 Fragmentation

Partitioning deals with the fragmentation of data entities. Most of the work on data fragmentation has been developed for textual and numeric data in relational databases (Ceri et al 1983; Ozsu and Valduriez 1991b), in other words, for *attribute* data.

When dealing with geographic entities, both the spatial and attribute components must be considered in the fragmentation (Veenendaal and Hudson 1992). For attribute data stored in relational tables, horizontal and vertical fragmentation strategies can be used[1]. However, such fragmentation cannot be performed in isolation from the spatial data that is often stored in a separate database and linked to the attribute component. In addition, topological relationships must be maintained in any fragmentation of a topological vector database. The spatial entity instances and their interrelationships must be retained for applications that query them. One solution is to replicate the entities that fall across the boundary of two or more fragments (Patel *et al.* 1997). An alternative is to split the spatial entity instances across the fragments, in effect generating additional entities.[2] In order to handle spatial data, additional fragmentation strategies, referred to as *spatial fragmentation*, are necessary. Horizontal, vertical and spatial fragmentation can be combined to provide a *mixed fragmentation* solution for geographic data. These strategies are detailed in the sections that follow.

[1] See section 2.4.2.2 for horizontal fragmentation and section 2.4.2.1 for vertical fragmentation.
[2] This is further explained with examples in Sections 2.4.2.3 and 2.4.2.4.

In determining the correctness of a fragmentation, the following rules are defined (Ceri and Pelagatti 1984; Ozsu and Valduriez 1991b):

1. *Completeness.* If an entity E is decomposed into fragments E_1, E_2, ...E_n, then each entity instance e_i that is found in E must also be found in one or more E_j's. This ensures that there is no data loss in the fragmentation process.

2. *Reconstruction.* If an entity E is decomposed into fragments E_1, E_2, ...E_n, then it is possible to reconstruct E from the E_i's.

3. *Disjointness.* If an entity E is decomposed into fragments E_1, E_2, ...E_n, then an entity instance e_i that is found in fragment E_j is not found in any other fragment E_k $(k \neq j)$.

For geographic data these rules apply to the spatial and attribute components as well as to their interrelationships. The *completeness* rule ensures that all points, lines, polygons, relationships and attributes are retained in the fragmentation. The second rule concerning *reconstruction* ensures that these fragments can be combined to reconstruct the original entity. For spatial data, this is accomplished with a *union* spatial overlay operation. The *disjointness* rule avoids data redundancy among fragments. For vertical and spatial data fragmentation this rule cannot always be strictly maintained. In particular the spatial fragmentation of topologically encoded lines and polygons may result in nodes and lines being replicated so that the spatial entities can be maintained in individual fragments. For vertical fragmentation, the key fields are replicated across fragments so that entities can still be accessed.

2.4.2.1 Vertical fragmentation

Vertical fragmentation involves fragmenting an entity based on the usage of the attribute fields (*eg*. the columns in a relational table). One or more fields that are accessed together with a high probability can be vertically separated from the other fields. Some replication is introduced since each new table must duplicate the primary key required to facilitate access. For this reason, as mentioned in the previous section, the *disjointness* rule of fragmentation can not be strictly maintained.

A GIS attribute table can be vertically fragmented resulting in two or more new tables. All tables would have identical primary keys and would refer to the same geographic entities. Figure 2.12b indicates an example of a vertical fragmentation of the attribute table of the dataset of Figure 2.12a.

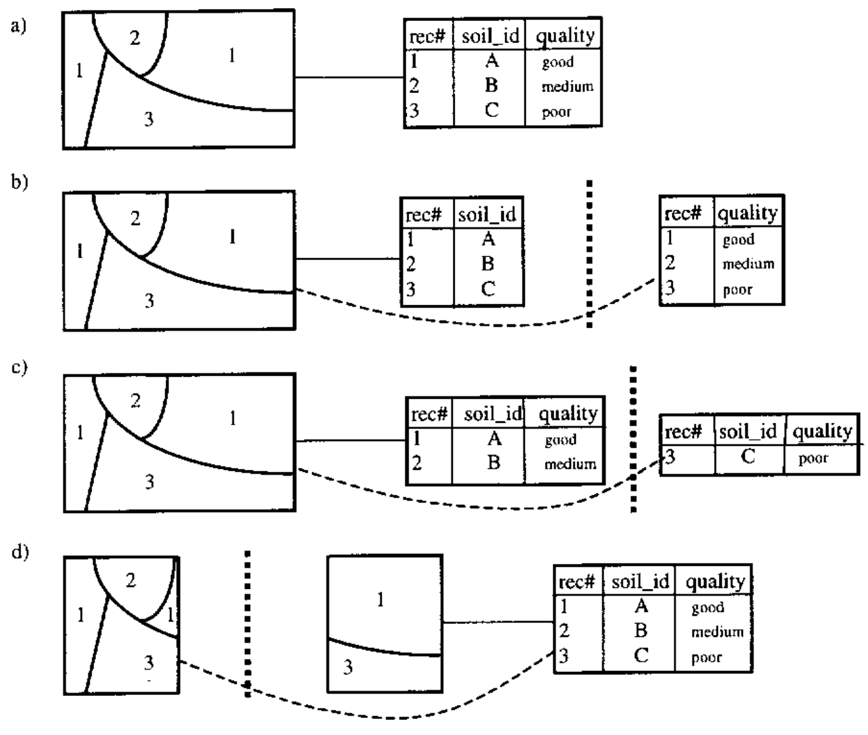


Figure 2.12 Types of data fragmentation: a) the original geographic dataset, b) vertical fragmentation, c) horizontal fragmentation and d) spatial fragmentation

Ceri and Pernici (1985) addressed the vertical fragmentation problem by considering the usage of the attributes of a relational table. Different operations may access different and not necessarily disjoint sets of attributes for the same entity. A potential vertical fragmentation is evaluated for two disjoint sets of attributes of an entity by determining the number of accesses requiring only one of the two fragments and the number of accesses that require both fragments. If their difference is positive, then a vertical fragmentation is potentially advantageous.

An early use of vertical fragmentation was to improve access to data in memory. The attributes of data that are most heavily accessed were stored separately from those attributes least accessed (Hoffer and Severance 1975). Such fragmentation was designed for single-site systems and involved fragments that could be overlapping. Navathe *et al.* (1984) later extended this work to include non-overlapping fragmentation for distributed database systems where multiple networked sites are involved.

The approach used by Navathe *et al.* (1984) is a top-down methodology involving the splitting of an entity (Ceri and Navathe 1983; Navathe and Ceri 1985; Ozsu and Valduriez

1991b). The attributes of an entity are initially grouped together as a fragment and then split as new fragments are generated. Such an approach is in contrast to the bottom-up methodology where the individual attributes initially form separate fragments that are then grouped to obtain the final vertical fragmentation. The *splitting* approach is preferred to the *grouping* approach because the final solution is much closer to the set of all attributes than to sets containing single attributes (Navathe *et al.* 1984).

The splitting technique for vertical fragmentation requires information such as individual attribute usage for each transaction, site accesses per time period and size of each attribute in the record. Navathe *et al.* (1984) includes the type of access to distinguish between a *retrieval* and an *update* access. An *attribute affinity matrix* is generated and shows the frequency of accesses of attributes accessed in conjunction with each of the other attributes. An iterative binary partitioning algorithm, referred to as the *Bond Energy Algorithm (BEA)* and developed by McCormick *et al.* (1972), is used to cluster the matrix so that large values are grouped with large values and small values are grouped with small values (Navathe *et al.* 1984; Ozsu and Valduriez 1991b). A splitting algorithm is employed on the clustered affinity matrix to generate non-overlapping fragments by selecting a point on the diagonal of the clustered affinity matrix to separate the top-left cluster from the bottom-right cluster. The algorithm is recursively applied until the desired level of fragmentation is obtained.

Cornell and Yu (1987 and 1990) extend some of this work and develop a binary partitioning algorithm that applies vertical fragmentation to improve disk accesses for a relational database. The algorithm takes into account the specific query access plan chosen by the query optimiser and minimises the average number of disk accesses. Similar techniques have been adopted by Oracle to improve the performance of join operations in their relational database (Oracle 1999). Vertical fragmentation results in significant performance improvements particularly for disk input/output bound situations.

Rather than use the binary partitioning algorithm, Navathe and Ra (1989) have implemented a graph algorithm which identifies partitions from a graph. The affinity matrix is used to construct an *affinity graph* from which affinity cycles are identified to form the basis of vertical fragments. This algorithm can be used for both single-site memory level partitioning and multiple-site distributed data partitioning.

2.4.2.2 Horizontal fragmentation

Horizontal fragmentation occurs when instances of an entity are fragmented based on one or more attribute values (*ie.* rows or tuples are fragmented). Figure 2.12c indicates horizontal fragmentation of the attributes of the geographic dataset shown in Figure 2.12a. The attribute records have been fragmented into two different tables. For geographic data each attribute record is linked to one or more spatial features while each spatial feature is linked to only one attribute record.

Ceri and Navathe (1983) developed an algorithm for horizontal fragmentation by considering all candidate partitionings of a given entity (see also Ceri and Pernici 1985; Navathe and Ceri 1985). Each candidate partition is identified with a list of predicates on the given entity. All possible conjunctions of predicates (one from each candidate partition), referred to as *minterm fragments*, are then identified. The authors recognise that the computational complexity of obtaining minterm fragments is very high. The number of accesses for each site are evaluated for each minterm fragment which is then allocated to the site associated with the highest number of accesses. When the fragmentation is completed the allocation of fragments is used as the starting point in a post-optimisation step to introduce redundancy. The database designer uses operation frequency information to iteratively identify candidate redundant allocations and evaluate them using the same criteria as for fragmentation. The last step is to take the allocation of fragments and produce the local schema for each site.

Ozsu and Valduriez (1991b) detail an algorithm for defining minterm predicates for a relation given a set of simple predicates. These minterm predicates refer to horizontal fragments and are derived from information on how the application accesses the relation. The algorithm assumes that it is possible to identify the access probability of each tuple for each application under consideration and that relevant simple predicates can be identified. While the determination of the minterm predicates is straightforward the computational time involved is not, since the set of minterm predicates can be quite large, in fact, exponential on the number of simple predicates.

The last part of the algorithm involves eliminating minterm predicates that are conflicting (*ie.* would result in an empty fragment) and are therefore meaningless. The authors do not specify how relevant simple predicates and meaningless minterm predicates are to be identified. Because this requires a considerable amount of work, even if it is possible to obtain such information for all the applications, the authors, rather, appeal to the common sense and experience of the database designer.

Two types of horizontal fragmentation are identified by Ozsu and Valduriez (1991b): primary and derived. A primary horizontal fragmentation is defined by a selection of tuples on a relation while a derived horizontal fragmentation is defined on another relation related to the first relation with a join operation. Section 2.5.1 further discusses primary and derived predicates.

2.4.2.3 Fragmentation for geographic data

Fragmentation is not a new issue for geographic data since it occurs already in the construction of analog maps as well as digital databases. Tiling methods and spatial indexing techniques use fragmentation to improve storage and access of information. These fragmentation strategies are discussed before additional ones for data distribution are considered.

The traditional means of geographic data storage, the analog thematic map, is a form of fragmentation. The division of data across several maps is determined by thematic information and the level of detail required by map users. For example, one map may contain land use and planning zone boundaries while another contains topography, both at a level of detail of a 1:75 000 cartographic scale (Figure 2.13). This essentially represents a vertical fragmentation. Further fragmentation is evident in that these maps are tiled into map sheets (Goodchild 1989). The level of tiling is dependent on the scale of detail. The example of Figure 2.13 depicts a thematic map tiled into 1:25 000 scale map sheets. Tiling is a form of horizontal fragmentation.

These forms of fragmentation have been extended into digital geographic databases. Tiling need not be concerned only with a regular rectangular partitioning, but irregular tiles based on administrative or other units can be used (as in the spatial *Librarian* system in ArcInfo (ESRI 1989)). Fragmentation, either overlapping or disjoint, is also used in spatial indexing methods to organise and/or access data on local storage devices (Frank 1981; Frank 1988; Noronha 1988; Li and Laurini 1991; Droge and Schek 1993). Some indexing methods, such as those based on a region quadtree (Samet 1990), use a regular fragmentation across the data, while disregarding the structure of the data, to determine each entry in the index. Other indexing methods such as the R-tree (Guttman 1984), K-D-B tree (Robinson 1981) and Cell tree (Gunther 1988) use the structure of the data (data-driven) to determine each entry in the index.

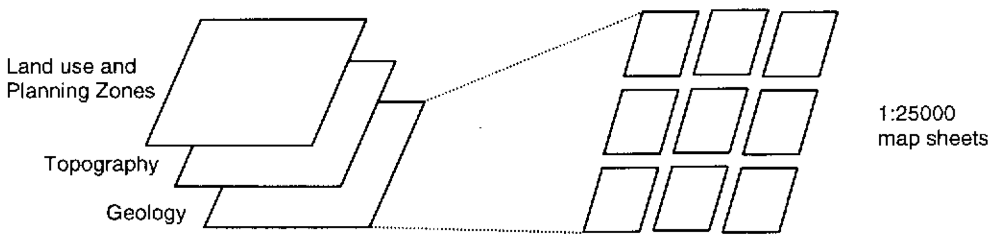


Figure 2.13 Fragmentation of analog maps

Whereas many of the fragmentation strategies for spatial access methods are data-driven, Droge (1994) presents a query-adaptive method called *Patchwork* that dynamically determines storage *clusters* suited to a set of spatial queries. The clusters are essentially spatial fragments based on irregular rectangular *patches* that are disjoint and cover the entire data space. Extended objects that span patches are clipped and stored as separate parts in separate clusters, thereby avoiding object duplication. Spatial objects are also clipped at query range boundaries. A user query is constructed by using an index directory to obtain the relevant clusters and by checking all objects and parts of objects for intersection with the query. The cost involves a *composition* cost for reconstructing an object from its parts and a *false-drop* cost associated with discarding unwanted objects or parts of objects. Heuristic algorithms are used to maintain a balance between a fine and coarse fragmentation. This work does not address attribute data but only addresses spatial data and its fragmentation.

Although Droge's work is focussed on indexing the concept is adapted and used for data fragmentation in this research. The query-adaptive approach, although more complex, lends itself to a changing data usage environment. Most current GIS are either single site or use a replication strategy. A data-adaptive approach to the distribution of data would consider the structure of the database. A query-adaptive method would additionally consider the queries and associated usage of the dataset. The advantage of such an approach would be improvements in efficiency of data access and reduction in data redundancy, since the data can be located to cater for the data access requirements of particular queries.

In this research fragmentation is being used for the purpose of data distribution. A number of strategies can be considered:

> ***Random*** - Entity instances for all entity types are randomly assigned to fragments. The number of fragments required must be determined prior to the assignment. This scheme is by far the simplest since no information regarding the data or the applications needs to be considered.
>
> ***Tiling*** - Data space is divided, using spatial and/or attribute properties, into *tiles*. Entity instances that lie within a particular tile are all assigned to the same fragment. The tiling may be regular or irregular. For example, a regular tile for the attribute data space could be an equal-intervals classification, or a rectangle (range of coordinates) for the spatial data space. A irregular tile could be based on an administrative unit (spatial) or on an attribute such as a "time period" (*eg*. occurrences between 1988 and 1998). Such a scheme is relatively straightforward and only uses information about the data properties. However, because it does not

consider the frequency of distribution of entity instances among the tiles, the fragments may be unevenly balanced with some containing almost all of the data and others containing very little or none. Further, no information about the applications that use the data is considered.

Quantile - Given some ordering on an entity, based on either spatial or attribute properties, the entity instances are assigned to fragments with an equal number of instances in each fragment. This scheme does consider the distribution of instances within the data space and equally divides the data among the available fragments. Chang and Cheng (1980) use a hashing technique to uniformly distribute data values of a relation among fragments. Whang *et al.* (1994) uses a nested approach where a directory is used to maintain a nested rectangular partitioning of the domain space. A lower level of the directory provides a finer granularity of fragmentation and a higher one coarse granularity. Once again, however, information about the applications is not considered in this approach.

Usage-based - Since applications do not normally access a database *uniformly*, it would be sensible to distribute data among fragments based on usage properties. Recall from Section 2.2.4 that *data usage properties* are low-level factors that contain information useful for data distribution. Rather than have one fragment with all the high-usage data being accessed frequently, the high-usage data could be dispersed among the fragments to avoid data access bottlenecks occurring. Such a strategy will involve more complex methods such as horizontal, vertical and spatial fragmentation (see Section 2.4.2.4) which are based on the usage properties of entities and instances.

Of these methods, the *Tiling* and *Quantile* methods are data-adaptive, the *Usage-based* method is query-adaptive, and the *Random* method is neither. Because of the advantages of query-adaptive methods, as discussed above, this dissertation focuses on *usage-based* fragmentation strategies. For geographic data where the spatial and attribute components are stored together in a relational structure, horizontal and vertical fragmentation methods can be used. However, these methods do not consider the spatial properties of geographic data. In a fragmentation spatial entities have to be maintained by either replicating or adding data (Kriegel *et al.* 1991).

For example, consider a spatial entity represented as a polygon. The polygon could be stored across three tables, a *polygon* table indicating which lines make up the polygon boundary, a *line* table indicating the nodes (*ie.* end points), vertices, and topology of the lines, and a *points* table providing the coordinates of the nodes (Laurini and Thompson 1992). Treating this polygon as a fragment of a larger dataset would involve identifying one or more records in each of the three tables, these records being interdependent. Further, this polygon may be associated with other neighbouring polygons (*ie.* common lines and nodes). In order to maintain the polygons in a fragmentation, the common lines and nodes would be duplicated in each fragment. In a topological vector database, such duplication would also be necessary to maintain topology in each fragment. This is different from horizontal fragmentation that deals with each record as an independent atomic unit and is more akin to the primary key repetition found in vertical fragmentation.

A further example involves the fragmentation of a polygon. Such a fragmentation may result in separating the points or lines of a polygon boundary and placing them in two different fragments. However, the concept of "polygon" would be lost since only portions of the polygon are found in each fragment. Maintaining these portions as complete polygons is essential if they are to be used as individual entities in different fragments. Therefore, for the purpose of fragmentation, the one original polygon may have to be split into two separate polygons that are placed in separate fragments. In this process, the two resulting polygons will have to be reconstructed by adding or replicating points and lines to form a complete boundary. This is further explained in Section 2.4.2.4.

As already stated in Section 2.4.2, because of the need to maintain spatial relationships and because a fragmentation may result in additional spatial data being generated, the traditional horizontal and vertical fragmentation strategies are not sufficient in themselves. Rather, they must be extended with a *spatial* fragmentation strategy for geographic data.

2.4.2.4 Spatial fragmentation

Spatial fragmentation strategies are used to fragment spatial data according to various spatial and/or attribute criteria. This fragmentation can be based on specific properties of the spatial dataset, some pre-determined fragmentation such as a regular tiling or some mapsheet boundary, or some other spatial boundary or feature from another dataset. In conjunction with spatial fragmentation, horizontal and vertical strategies, based on spatial and/or attribute criteria, can be used on the corresponding attribute data.

For raster data, consisting of a regular tessellation of cells, spatial fragmentation can take advantage of the regular size and shape of each cell and its neighbouring cells. Fragmenting the data space based on nested or hierarachical tessellations such as quadtrees or using various cell orderings such as the Morton order are already used for spatial indexing and storage of raster data (Orenstein and Manola 1988; Samet 1990; Laurini and Thompson 1992).

Point datasets are the simplest vector data types to fragment since each spatial object is a single atomic point representing an instance of an entity. Spatial fragmentation of the entity is a matter of allocating each separate point to a fragment. There are no explicit topological relationships that need to be maintained and hence this fragmentation is akin to the horizontal strategy.

Whereas point features are topologically independent and not related by adjacency, lines can be adjacent and share common nodes. Fragmenting line data, such as rivers or contours, may not only involve splitting up an entire data entity (*eg*. an individual river or contour), but may also involve the splitting of adjacent lines at their common node, or splitting up of line segments themselves (Figure 2.14a) resulting in the creation of additional nodes.

The fragmentation of polygon data is more complex since polygons share common lines and nodes. These lines and nodes may have to be replicated in the various fragments (as described by Patel *et al.* 1997) or the lines may have to be split to create additional nodes and lines as illustrated in Figure 2.14b. This may result in generating multiple polygons from one original polygon and allocating the newly generated polygons to different fragments. Access to the original polygon requires reconstructing it from the associated individual polygons in each fragment. The creation and management of additional data increases the complexity of such a fragmentation.

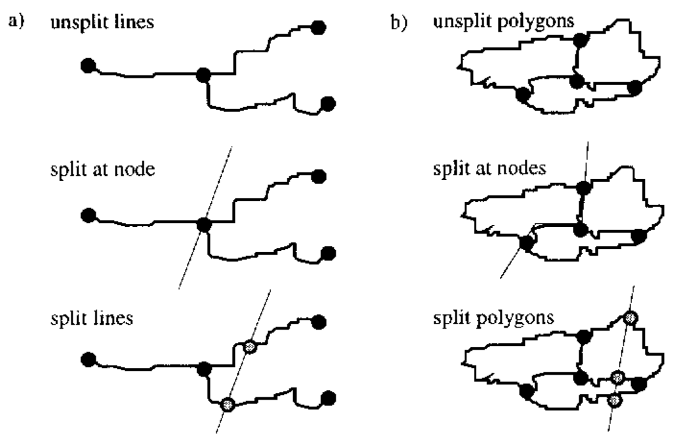


Figure 2.14 Fragmenting lines and polygons

2.4.2.5 Mixed fragmentation

Very little research has been done for simultaneously handling multiple fragmentation methods. Ra and Park (1993) describe a hybrid vertical and horizontal fragmentation technique using the graphical technique of Navathe and Ra (1989) for vertical fragmentation. Both horizontal and vertical candidate fragments are identified and placed in a grid. The grid is then optimised by merging grid cell fragments based on their access frequencies obtained from the most important application transactions. The remaining mixed fragments are then allocated to sites based on the usage information.

A combination of not only horizontal and vertical strategies but also spatial strategies can be used to fragment geographic data. Both the spatial and attribute components of geographic data should be considered in the fragmentation process. For example, it would be quite common to spatially fragment the spatial component and horizontally fragment the corresponding attribute data to keep the spatial and attribute components together in the same fragment (Figure 2.15).

If there is a one-to-many relationship between attribute instances and spatial entities, then replication must occur in the fragmentation process. For example, if a particular attribute is associated with two polygons placed in separate fragments, then the attribute must be replicated in each fragment. This occurs in addition to any line and node replication caused by adjacent polygons in separate fragments.

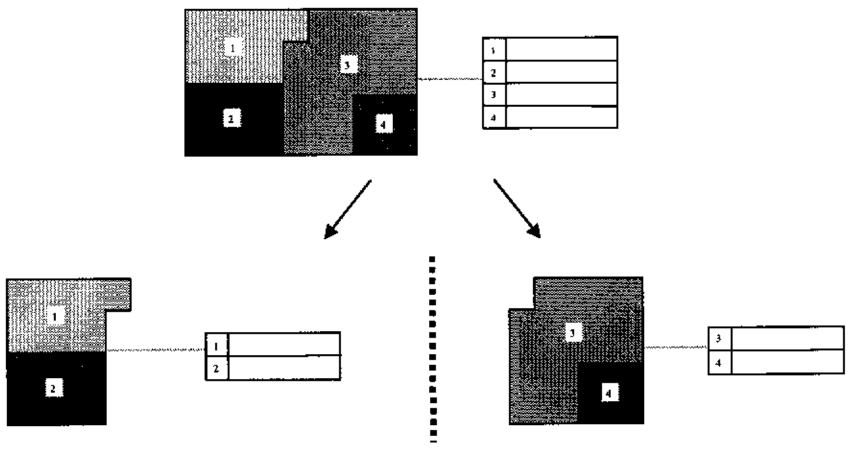


Figure 2.15 Spatial and horizontal fragmentation to retain spatial-attribute linkage

It is possible that the spatial and attribute data are accessed independently. In such a case the spatial and the attribute data must be dealt with separately in the fragmentation and may be allocated to different fragments. No replication need occur among the attribute data. Any access to the entity involving both the spatial and attribute data requires retrieving the two components from different fragments. The link between the two components must be maintained so that the entity can be reconstructed.

GIS handle the link between the spatial and attribute data in various ways. The ArcInfo GIS handles it by using an identification number (ID) which relates a spatial entity (*ie*. point, line or polygon) to an *attribute table* (ESRI 1994). This attribute table, which may be a point, line or polygon attribute table, has a one-to-one relationship with the spatial data primitives themselves and can then be linked to other user-defined attribute tables on a one-to-one or one-to-many basis. In such a case, spatial fragmentation must consider both the spatial data and the *attribute table* associated with it (*ie*. a polygon attribute table related to polygons). Because of the one-to-one relationship, no replication resulting from the fragmentation is necessary. The user-defined attribute tables can then be handled separately in a fragmentation strategy that would consider their access characteristics apart from that of the spatial component.

2.4.2.6 Levels of abstraction

When considering data distribution for GIS it is important to determine at what level of granularity the fragmentation takes place. Should fragmentation occur at a low level of abstraction among the simple spatial primitives, or should it occur for the spatial data entities at a higher level of abstraction?

At the internal level of a data model (see Section 2.2.2) vector geographic data are stored as simple spatial primitives in the form of points, lines, and polygons. Complex spatial primitives, such as dynamic segments (*ie*. complex lines) and regions (*ie*. complex polygons), can be constructed from one or more of these spatial primitives. At the external level, users and applications deal with spatial entities which themselves are constructed from complex or simple spatial primitives. The level of abstraction increases up the pyramid as shown in Figure 2.16. Note that the components of the data must be maintained from the external level through to the internal level in a geographic database.

To illustrate with an example, an application may deal with a network of roads. A *road segment* between two intersections could be represented as a simple spatial primitive (*ie*. a line). A *street* could be represented as a number of these road segments (whether connected or not) which would form a complex spatial primitive. Multiple complex primitives representing streets could be associated to form a spatial entity such as a *route*. Note that some spatial entities can be simple rather than complex (*eg.* a route is made up of a line segment).

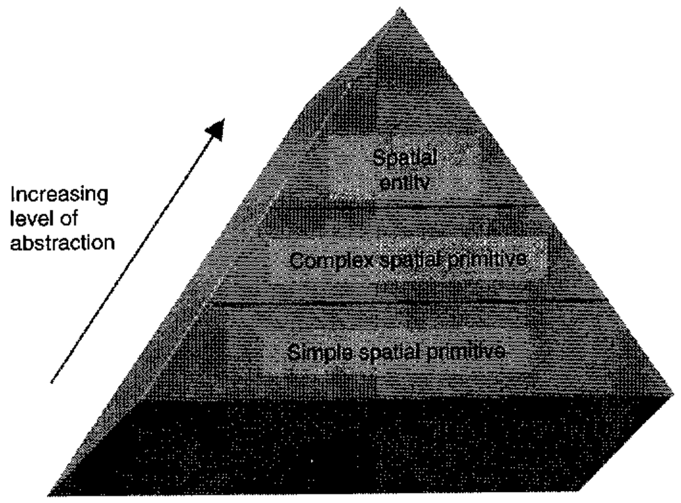


Figure 2.16 Levels of abstraction for geographic data

Part of the data distribution problem, then, is to determine at what level of abstraction the fragmentation of data should be considered. Applications access data at the external level. Fragmentation must occur at the internal level where the spatial primitives are stored. Therefore the problem, addressed in this dissertation, becomes one of translating usage information at the external level of a data model to a data distribution at the physical level.

2.4.3 Allocation

The allocation of data to sites is not an arbitrary decision but could be based on the usage pattern of the data. The data partitions can be constructed from data that is related by usage. Information on data *locality* can then be used to allocate the partition to the site "closest" to where it is being used or primarily being used. In determining a data allocation consideration is to be given to the costs involved in accessing and maintaining partitions on various sites. Such costs may include the cost of transfer, processing, and storage of partitions.

Most of the early data allocation models do not include the dependence between data but rather assume that data fragments are placed independently of each other. In reality, data fragments are often used in conjunction with each other by operations within applications and their interrelated usage patterns will affect their distribution. Apers (1982) uses a graphing strategy and heuristics to assign both data fragments and operations (queries and updates) to sites in order to minimise data communications costs and response time. In such a model the dependencies between data partitions are recognised, based on how they are accessed by user applications.

2.4.4 Replication

Data replication involves determining the number of data copies, if any, that should be placed among the sites of a distributed network. The purpose of replication is to improve performance of access to data by multiple applications on different sites and to improve reliability of access in the event of site failures (Andleigh and Gretzinger 1992). Replication also provides parallelism by allowing simultaneous access to different copies of the same data stored at separate sites (Wong and Katz 1983; Hevner and Rao 1988).

Replication of data in a distributed network is a result of allocating multiple copies of partitions to different sites. Access is often required for only some of the data in a partition, in particular for the cells which make up the partition. Thus, rather than replicating the entire partition, it may be more efficient to assign copies of cells to separate partitions. Such a placement would be determined by analysing data access patterns for each cell rather than for the partition as a whole.

Replication may also be required to maintain information during the partitioning process. This has already been explained in Sections 2.4.2.1 and 2.4.2.3.

A disadvantage of replication relates to the cost of updating the data copies and maintaining consistency among these copies. If access to the data involves only retrieval, then of course there is no further cost and the replication may indeed be beneficial. When a copy of the data is modified or updated, then it is important to ensure that replicates of the data reflect these changes. Related to this are the problems of providing concurrency control, integrity of data, recovery from system failures and transaction abortions, and measures of security (Bernstein and Goodman 1981; Hevner and Rao 1988; Ozsu and Valduriez 1991a and 1991b). Replication is a complex issue and, other than the replication necessary to prevent information loss, is not dealt with in this dissertation.

2.4.5 Data migration

Data distribution occurs in a dynamic environment where data and applications change over time. This is particularly so within a GIS environment where applications are dynamic and varied (Coleman and Zwart 1992a; Goodchild and Rizzo 1986). Data distribution therefore cannot be a static process but must be monitored and modified to reflect changes regarding the access and usage of the data within the distributed environment.

An initial data distribution must therefore be followed by a data migration strategy (Figure 2.17). The initial distribution would normally be determined in the design stages of a system and applied as datasets are obtained in the implementation phase of an application. For an existing system, an initial (but often *ad hoc*) distribution is already in place but may need to change to reflect *current* usage of the data. Information obtained from the design of the data and operation schema as well as information regarding the predicted usage of the data can be used as input to the initial distribution design.

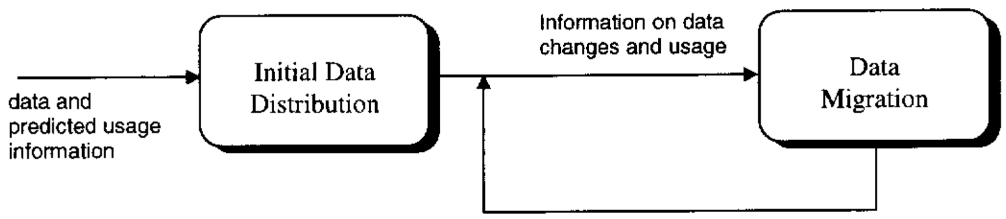

data and predicted usage information
Initial Data Distribution
Information on data changes and usage
Data Migration

Figure 2.17 Stages of dynamic data distribution

As further information about data usage and schema modifications becomes available, the dynamic data migration strategy can be used to redistribute the data. This is an iterative process that must be executed at appropriate intervals. The frequency of data migration is an important choice and must reflect the application environment. Too frequent migration means the overhead costs of redistribution may exceed the advantages and too infrequent migration may result in the data distribution not reflecting the application changes in data usage (Sacca and Wiederhold 1985; Stonebraker *et al.* 1996). An explanation of how data migration is incorporated into the data distribution methodology is presented in Section 4.2.4.

2.4.6 Database and application information

The information required to effect a data distribution varies widely, ranging from information on the overall structure of the data to details on the usage of data at each individual network site. Obviously the amount of information available will determine the effectiveness of the resulting distribution. A good distribution should adapt to a changing environment. However, this will involve a considerable amount of information about the database and applications being continually monitored, collected, analysed and acted upon.

Even if all the criteria necessary for a distribution design are identified, much information may not be available when it is needed. This is especially true when considering usage data.

Of course usage data are not available before the applications are run. Therefore usage data from previously-executed applications are gathered and used to predict how future applications will use the data. Ozsu and Valduriez (1991b) suggest that if it is not possible to analyse all of the user applications, one should at least investigate the most "important" ones. The *80/20 rule* suggests that as a rule of thumb, 80 percent of data accesses are initiated from 20 percent of the most active application queries (Wiederhold 1982). Collecting such information assumes that previously-executed applications indicate how future applications may use the data. Information that is available is better than no information at all as long as the user of that information understands its limitations.

Situations may arise where information such as usage data from previously-run applications are unavailable. Information concerning the entity schema and operation schema as well as any distribution requirements must be obtained during the requirements analysis stage. The *entity schema* identifies the structure of the database whereas the *operation schema* identifies the operations and the data they access. Other required information, such as operation frequencies and data access frequencies, will have to be predicted in the absence of any usage data. An initial data distribution can then be determined from the given information (Ceri *et al.* 1983; De Antonellis and Di Leva 1985).

The information required for distribution design can be categorised as database information and application information. Database information, sometimes also referred to as *metadata*, deals with the structure and size of the data whereas application information identifies how the data is used among network sites by applications and users.

Database information

For data distribution design both the structure and size of a geographic database must be known. Such information is obtained from the entity *schema* and from the conceptual design stage (*eg*. from an Entity-Relationship model) (Batini *et al.* 1992). For geographic entities, information must be provided for both the spatial and attribute components (Firns 1992). Figure 1.4 illustrates an Entity-Relationship diagram for the GEOMINE application. The entities and their attributes are identified. For spatial entities, the spatial data type (which is *point*, *line* or *polygon* for a vector database) is provided.

Further information relating to the size of the entities must be obtained from the physical design stage when the size of the spatial and attribute components is known. Size information is used to determine the volume and cost of moving data from one site to another.

Application information

Information on applications and how they interact with geographic data are more difficult to determine than for database information. It is expected in the design stages of an application that an initial estimate of application information is known so that an initial data distribution can be determined. Further information can be obtained later when the applications are executed.

The representation of an operation schema is difficult because of the potentially large number of operations and their complexity, particularly for spatial operations and data. Depending on the spatial operation, the spatial and attribute components may be treated differently and the number of spatial and attribute records accessed may be different. Ceri and Pernici (1985) use an *operation ER model* for non-spatial operations that includes a means of recording the number of instances of a record, how many records were accessed and the type of operation (keyed access, retrieval, output, write, etc.). For geographic data, an operation ER model must provide this information for both the attribute and spatial components. The number of instances recorded for spatial components is supplied using number of points (for the reason already discussed above for database information).

Figure 2.18a illustrates an Operation ER diagram (based on that of Ceri and Pernici 1985) for an operation "*located within*" which finds all mines located within the *Laverton* map sheet area. The *mapsheet_id* and *mapsheet_name* attribute fields in the *mapsheet* entity are accessed through an index on the *key* field (access type = K) and one field (where *mapsheet_name* = *Laverton*) read (access type R = retrieval) from a total of 98 instances of map sheets. Four (defining the 4 corners of the *Laverton* map sheet) of the 422 points which make up the map sheet polygons are accessed by this operation. Of the *mines* entity, 8333 records are accessed sequentially (access type = S) by *mine_id* and 503 records are selected. The *mine_name* attributes as well as the spatial locations for the selected mines are retrieved for output (access type = O). Note that in this case where the spatial data type is *point,* the number of spatial records accessed is equivalent to that of attribute records since each point has one attribute record associated with it.

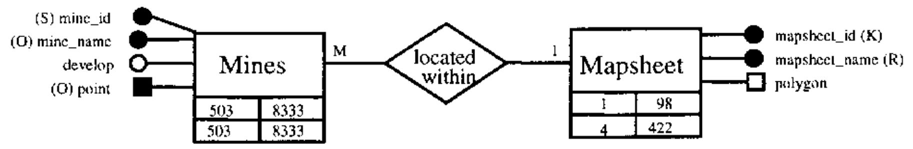


Figure 2.18 (a) Operation ER diagram for the operation: "*Identify all mines located within Laverton*"

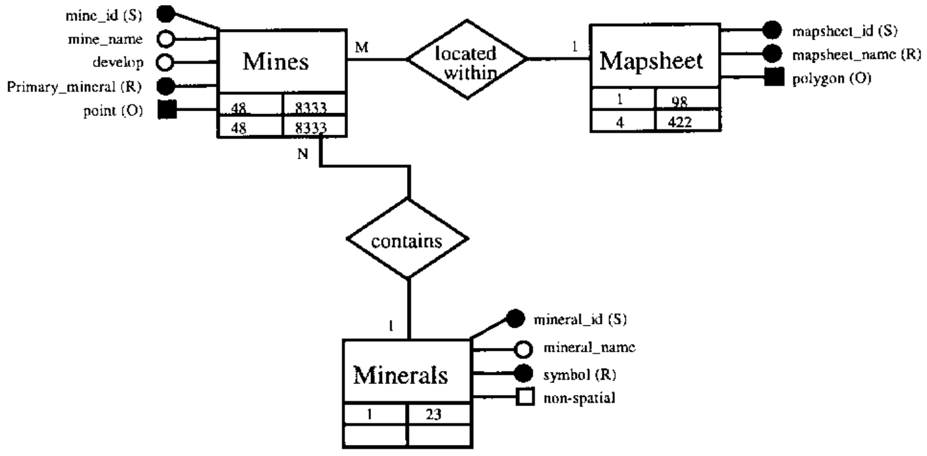


Figure 2.18 (b) Operation ER diagram for the operation: "*Display mines located within Laverton and containing 'Au' as primary mineral*"

Figure 2.18b illustrates an Operation ER diagram where the same operation is further constrained to gold mines. The *minerals* entity is accessed sequentially to retrieve the record where *symbol* = *'Au'*. From the mines entity, 48 mines entities are retrieved that match the selection criteria. Four line segments of the *Laverton* map sheet polygon and 150 mine points are accessed for output. The Operation ER diagram also shows that the *Laverton* map sheet is retrieved for output. In general, the information contained in the Operation ER model must be obtained for all operations across all applications.

Site information

Information about computer network sites is necessary to determine any requirements and constraints for placing data. Organisational policy may dictate that certain data must or may not be placed on a particular site. Further, the storage constraints must be known for each site. Such information can be identified by a Database Administrator (DBA) and must be updated as requirements and constraints change (see Sockut and Goldberg 1979 for issues that the DBA must consider).

2.5 Predicates and Selections

Applications that access an entity often do not require the whole entity but simply a fragment of it. This fragment can be a spatial subset referring to the data within a window defined over a geographic area or it can be an attribute subset referring to only the data with the specified properties. For every application, there must exist a means of specifying the fragment of data required by the application operations. This is achieved using various operations that incorporate the use of predicate expressions to specify the data. These operations are often based on the fundamental SQL relational algebra operators: select, project and join (Date 1982; Aho and Ullman 1992; Korth and Silberschatz 1991). The predicate expressions and the associated selection operations are used in the data distribution design methodology detailed in this dissertation.

To illustrate the use of predicates and selections, the following selection example, based on the relational algebra "select" operator, contains an expression identifying a subset of an entity called *mines*.

```
select * from mines where (mineral = 'Au') and (production > 0)
```

(The expression "*(mineral = 'Au') and (production > 0)*" is used to specify the condition that only mines which are producing gold are being accessed.) The condition is referred to as a *predicate* (*ie*. horizontal fragmentation).

Rather than select all attributes for entity instances that satisfy the condition, it may be desirable to select only a subset of attributes. This can be achieved using a projection operation to identify the attributes to be retrieved. For example, the above "select" operation could be extended as follows.

```
select mine_name,production from mines where (mineral = 'Au') and (production >
0)
```

In this case, only the attribute fields *mine_name* and *production* are retrieved for the instances identified by the given predicate. This projection essentially defines a vertical fragment of the entity *mines.*

The selection and projection operations indicated above operate on one entity. A join operation provides a means of relating entities together. The example above can be extended to include a join operation as follows.

```
select mines.mine_name, minetype.type_name from mines,minetype
        where ( mines.type_id = minetype.type_id) and (mines.production > 0)
```

Here the entity *mines* is joined to the entity *minetype* by the field named *type_id* from each entity. A selection operation (using the predicate "*mines.production > 0*") and a projection operation (identifying attributes *mine_name* from entity *mines*, and *type_name* from entity *minetype*) further specify the fragment of data being selected.

Other operations for selecting data exist and include the union, intersection and difference set operations between entities. These three operations together with the selection, projection and join operations provide a foundational basis which applications can use to select entities and subsets of entities. The predicates and fields specified within these operations can be used to identify the fragments of data being accessed by user applications. Because these operations are used to select subsets of entities, they will be collectively referred to as *selection operations* (not to be confused with the selection operation in relational algebra) for the purposes of this dissertation.

The operators and operands of an expression can be represented by a tree structure which is sometimes called an *expression tree* (Aho and Ullman 1992). The tree specifies a way of associating operands with operators and relating the operators together in an expression. This concept is used in the Predicate Fragmentation tree outlined in Chapter 3.

The selection operations can be implemented in various ways. As an example, consider the GIS which is used in this research (ESRI 1992; ESRI 1994). ArcInfo uses the INFO database management system which pre-dates the SQL standard. INFO implements three operations known as "reselect", "aselect" and "nselect", which further restrict a selection, add to a selection, and negate a selection, respectively. Within the first two of these operations, a predicate may be specified. A "relate" provides the information for joining two tables and can be specified within the predicate using the "//" operator. Further operations are available to insert and delete record instances, as well as insert, delete, and retrieve (selected) attributes of an entity. All these operations can be used together by an ArcInfo application to access subsets of a database in a manner equivalent to the SQL operations described above. Because ArcInfo is used in this research, the three INFO operators are used in examples throughout the dissertation.

A predicate identifies a subset of an entity that is being referred to. These predicates are specified using rules that combine operators and operands. The following Extended Backus Naur Form (EBNF) syntax describes, in some detail, how predicates can be constructed in ArcInfo:

```
predicate = simple_predicate | compound_predicate
simple_predicate = relational_expression | boolean_constant | boolean_function
compound_predicate = {simple_predicate} boolean_operator simple_predicate
relational_expression = relational_operand relational_operator relational_operand
relational_operand = local_field | foreign_field | constant
```

The constants, functions, and operators are further defined according to the rules of a specific GIS database management system. For example, the following definitions for boolean operators, relational operators and foreign fields would apply for ArcInfo:

```
boolean_operator = "AND" | "OR" | "NOT"
```

```
relational_operator = "<" | ">" | "<=" | ">=" | "<>" | "IN"
foreign_field = relate_name "//" [ local_field | foreign_field ]
```

In this example, *relate_name* is the name of a *relate* (*ie.* a relation defined between two tables), whereas *local_field* and *foreign_field* are the names of fields (attributes) in a local and foreign entity respectively, through which the join is performed (see below in Section 2.5.1 for explanations of *local* and *foreign*). A relate refers to the information that specifies how a logical join operation is to be performed. A relate is first defined and then referred to in subsequent operations which require a logical join.

A predicate is often based on one or more properties of the entity. In the example of the relational algebra selection operation given above, the mines are selected based on their attributes: mineral type (*Au*) and production rate (>0). This is essentially a horizontal fragmentation. For geographic data the selection can also be based on spatial properties. For example, the following predicate indicates that only *mines* found north of 32 degrees latitude are to be selected.

select mines where Y-coord > 32.00

Here the selection is based on a spatial property, the Y-coordinate value. A spatial query may also involve a spatial operation. The following example selects all mines in the 'Laverton' mapsheet.

select mines where mines.XY_coord $\cap$ *(mapsheet.mapsheet_name = 'Laverton')*

The operator "$\cap$" represents a spatial intersection overlay operation. These selections, involving spatial properties, essentially result in a spatial fragmentation on the mine points in addition to a horizontal fragmentation for the mine properties.

The selection operations and the predicates that they apply to within an application form the basis by which usage information is obtained and from which a fragmentation can be determined. They are crucial to the data distribution design and methodology outlined in this dissertation.

2.5.1 Primary and derived predicates

The *primary predicates* of an entity are based on the properties of that entity, whereas *derived predicates* are based on the properties of other related entities. The entity for which the predicate is being defined is referred to as the *local* entity whereas a related entity is referred to as a *foreign* entity. For geographic data, spatial and/or attribute properties of both local and foreign entities may be used in the predicate expressions. Table 2.1 lists some examples of primary predicates defined for various entities.

Table 2.1: Examples of Primary Predicates

	Entity	*Predicate*

a)	mapsheet	mapsheet_name = 'Laverton'
		mapsheet_name = 'Albany'
		NOT (mapsheet_name IN {'Laverton', 'Albany'})

b)	mine	status = 'Developing'
		status = 'Abandoned'

c)	mine	Y-coordinate <= 32 Y-coordinate > 32

For the *mapsheet* entity the predicates are based on its attribute *mapsheet_name* (Table 2.1a). The *status* of a mine is used to define the predicates in Table 2.1b. Table 2.1c provides an example where a spatial property, the location of mine points above or below the 32nd parallel (degrees latitude), is used to define the predicates.

The predicates need not be defined on some property of the local entity, but can involve the property of a foreign entity that is joined via some attribute field to the local entity. Table 2.2 lists some derived candidate predicates for the *mine* entity. For example, the predicates in Table 2.2a are based on the *symbol* attribute of the *minerals* entity which is joined to the entity *mine* via the *mine_id* attribute field. In Table 2.2b the predicate for *mine* is based on a field in the *minetype* entity.

Table 2.2: Examples of Derived Predicates

	Entity	*Foreign Entity*	*Association*	*Predicate*
a)	mapsheet	mine	mine_id	symbol = 'Au' symbol <> 'Au'

b)	mine	minetype	mine_id	type = 'Underground'
				type = 'Bedrock'
				type = 'Surface'

c)	mine	mapsheet	location	XY-coord $\cap$ mapsheet_name = 'Laverton' XY-coord ! $\cap$ mapsheet_name = 'Laverton'

As with primary predicates, a derived predicate can be based on a spatial relationship. Table 2.2c shows an example involving the spatial location of the *mine* entity related to the spatial location of a foreign entity called *mapsheet*. The predicate involves the intersection ("$\cap$") of mine locations (XY-coordinates) with the 'Laverton' map sheet. In this case, a spatial overlay operation (*ie.* intersection) is required to obtain the fragment defined by the predicate. A selection involving such a predicate is obtained by executing a spatial overlay operation followed by the selection operation. In this case the overlay operation would result in attaching a map sheet name to each mine point by incorporating the *mapsheet_name* attribute. The subsequent selection operation would then simply use the *mapsheet_name* attribute to identify the required mine points.

2.5.2 Primitive predicates

A predicate expression is used to specify the subset of an entity. A predicate specified within an application query is referred to as a *user predicate.* Often different user predicates defined by applications on the same entity will result in selections that may not be disjoint. This happens, for example, when one query on an entity overlaps with another query. Figure 2.19 shows an example where there is no data in common with Query 3 and Query 1 (or Query 2) whereas the data identified by "B" are common to both Query 1 and Query 2. Clearly, user predicates do not necessarily identify unique and disjoint fragments.

For data distribution, we would like to identify fragments where all the data in the fragment are always accessed together in a homogeneous manner. In other words, we want to identify disjoint *atomic* fragments. Predicates that define such fragments are referred to in this dissertation as *primitive predicates.* As indicated in Section 2.4.2.2, they are also referred to in the literature as *minterm predicates* (Ceri and Navathe 1983). The data identified by "A", "B", "C" and "D" in Figure 2.19 are primitive predicates.

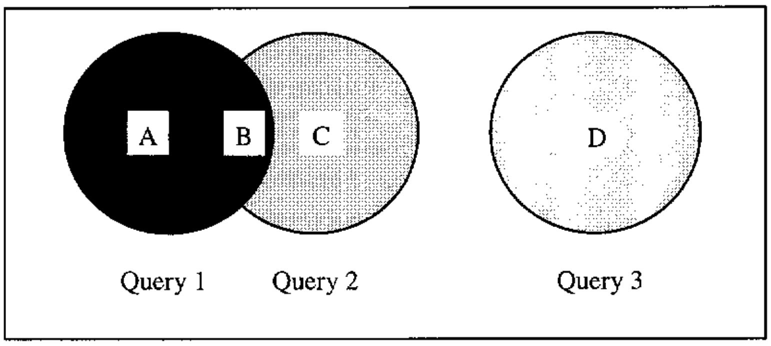


Figure 2.19 Venn diagram showing disjoint and overlapping data queries

To illustrate the concept of a primitive predicate with an example, suppose that an application selects some entities from an entity called *mines* based on the following user predicate expression.

mapsheet = 'Laverton'

Suppose that an application requests a selection on the entity *mines* based on a different user predicate expression.

type = 'Bedrock'

Suppose that the fragments defined by these two predicates are non-disjoint as illustrated in Figure 2.20. The data represented by group B is selected by both applications while the data represented by groups A and C are only selected by one of the applications. In addition, there exists a fourth fragment of data represented by D that consists of the data that was not selected for either of these applications. It is important to also consider this latter group of data since, according to the fragmentation rules specified in Section 2.4.2, all the instances of an entity have to be accounted for among the fragments produced by a data distribution.

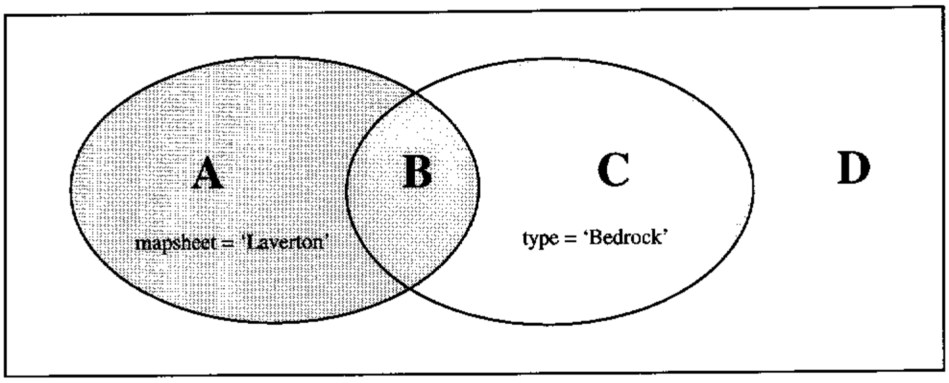


Figure 2.20 Venn diagram showing overlapping data selections defined by two user predicates

For the two user predicates shown in the example of Figure 2.20, four primitive predicates would result (defined by A, B, C and D). They are:

(Mapsheet = 'Laverton') and (type <> 'Bedrock')
(Mapsheet = 'Laverton') and (type = 'Bedrock')
(Mapsheet <> 'Laverton') and (type = 'Bedrock')
(Mapsheet <> 'Laverton') and (type <> 'Bedrock')

These four primitive predicates define four disjoint fragments and contain all the instances of data for the entity *mines*. Their union would result in the original entity. Thus, these primitive predicates define a fragmentation that adheres to all three correctness rules given in Section 2.4.2.

2.6 Methodology for Data Distribution Design

Ceri and Pernici (1985), De Antonellis and Di Leva (1985) and Ceri *et al.* (1987) describe a methodology for data distribution, used in their DATAID project. This project was undertaken with the objective of developing a methodology to support the specification, design and implementation of databases and tools for the designer. The distribution methodology involves two phases: analysis of distribution requirements and distribution design (Figure 2.21). The distribution requirements analysis is implemented in a step separate from that of the general requirements analysis step in the overall data design methodology. For the distribution requirements phase, the first task is to collect usage and frequency of access information and determine the distribution predicates based on both primary and derived predicates. For the second task these predicates are associated with quantitative values which identify their *polarisation*, that is the difference with respect to a homogeneous distribution of accesses. The partitioning tables identify the possible fragmentations derived from the predicates.

The distribution design phase uses the information gathered in the first phase to perform iterative horizontal and vertical fragmentation, redundant data allocation and the reconstruction of the logical schema at each of the local sites (Ceri *et al.* 1987).

The problems of horizontal partitioning, vertical partitioning, replication, and allocation have been independently addressed in the DATAID methodology. This is due to the complexity of modelling their interdependencies and solving for such a difficult problem (Navathe and Ceri 1985).

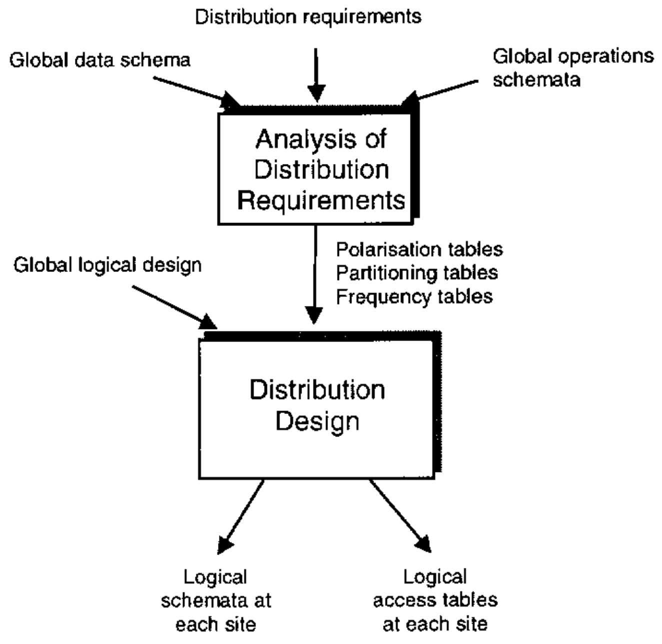

Distribution requirements
Global data schema
Global operations schemata
Analysis of Distribution Requirements
Global logical design
Polarisation tables
Partitioning tables
Frequency tables
Distribution Design
Logical schemata at each site
Logical access tables at each site

Figure 2.21 Data distribution design methodology (Ceri and Pernici 1985)

Hevner and Rao (1988) have used a data distribution design methodology similar to that of Ceri and Pernici. The distribution requirements are identified from the general requirements analysis step and analysed in a *distribution analysis* step. This step together with a *data allocation design* step make up the data distribution design stage in their methodology (Figure 2.22). The data allocation design involves data partitioning, placement, replication and dynamic migration. The data distribution design also includes a *local conceptual design* stage which uses the global conceptual schema and the allocation design to determine the local conceptual schema for each site.

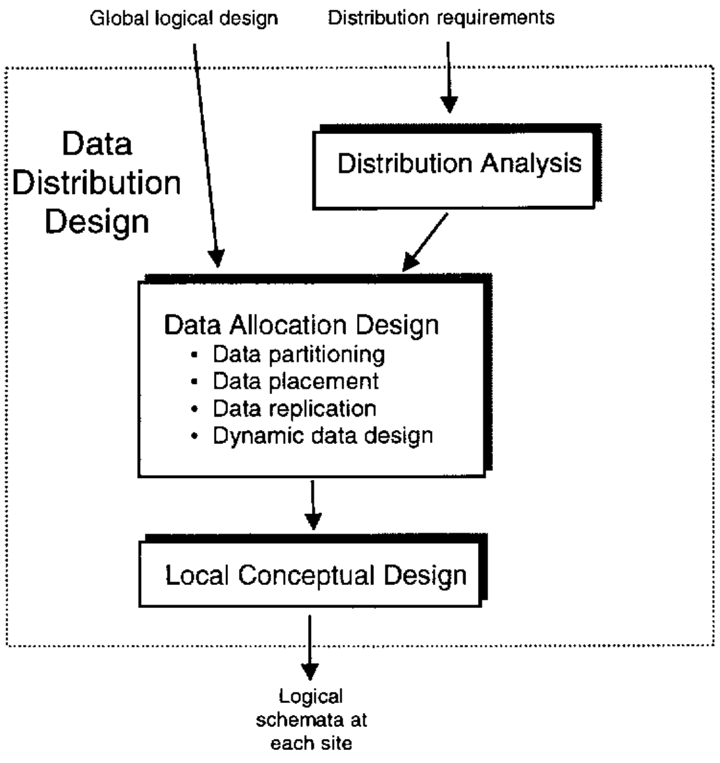


Figure 2.22 Data distribution design (Hevner and Rao 1988)

Both methodologies include a distribution analysis phase, to identify *what* the data and usage properties are, and a distribution design phase to determine *how* the data is to be partitioned and allocated based on these properties. These two steps are crucial and form the basis for the methodology used in this research and detailed in Chapter 4.

2.7 Conclusions and Problem Definition

This chapter provides the background information for the geographic data distribution problem. The distributed data environment was described and the development of the distribution problem was traced from simple file allocation to more complex data distribution. Many high-level factors affect data distribution but they are hard to identify, so low-level factors based on database and application information will be used to determine an appropriate distribution strategy.

The distribution design problem involves fragmentation, allocation and migration of data. A number of fragmentation and allocation solutions are outlined in the literature. However, they are oriented to non-spatial data and do not provide an integrated solution involving both entities *and* their interrelationships in a dynamic environment. Partitioning strategies for spatial data involve the duplication of existing primitives or the addition of new primitives to maintain spatial entities and topological relationships among the fragments. This research incorporates spatial fragmentation methods and aims to provide an integrated solution where a data distribution is *query-adaptive*, based on the dynamic usage properties of geographic data, rather than being simply *data-adaptive* and based only on the relatively static properties of the data. These usage properties are obtained from the predicates and data selections identified from application queries and are the basis from which a data distribution can be derived.

Having now defined the background material, the geographic data distribution problem can be more precisely stated as follows:

1. To develop and implement a model to store, access and maintain information required for the distribution of geographic data based on a topological vector structure. The PF model and its implementation are described in Chapter 3. The factors that affect the distribution, based on usage information, and the data fragments used in the distribution process must be identified from the PF model (refer to objectives 1 and 2 in Section 1.5).

2. To develop and implement a geographic data distribution methodology. The methodology must outline how the data fragments and usage properties are used to perform a data distribution. Chapter 4 outlines the methodology and its implementation (refer to objectives 3 and 4 in Section 1.5).

3. To develop procedures to evaluate the methodology. The testing procedures and measures are explained in Chapter 4 and the results are provided and discussed in Chapter 5 (refer to objective 4 in Section 1.5).

3 Predicate Fragmentation Model

3.1 Introduction

The predicate fragmentation (PF) model is designed and developed to identify geographic data fragments and determine their usage based on log information obtained from GIS applications. The basis of the model is a tree data structure defined for each entity set in the GIS database. Each tree is constructed from user-defined predicates that specify data selections from the database on which subsequent GIS *operations* are to be performed. Associated usage information is accumulated in the nodes of the tree. From the PF tree, primitive predicates defining cell fragments can be identified.

The tree can be pruned to eliminate predicates that are redundant or produce empty or undesired fragments. The resulting primitive predicates identify the fragmentation that forms the basis for data distribution. The PF tree is logically equivalent to an expression tree using the AND, OR, and NOT operators, but is more efficient in its implementation because it takes advantage of some simple heuristics.

3.2 Overview of Geographic Data Fragmentation

The geographic data fragmentation problem involves the use of information about application queries to obtain fragments from the entity sets of a GIS database (see Section 2.4.2.3). Because queries overlap, a perfect fragmentation cannot be obtained from the queries alone. Rather a fragmentation should be based on *cells* that identify disjoint fragments. The solution adopted here is a predicate fragmentation model (PF model) that provides a means of manipulating user predicates so that primitive predicates can be determined to identify fragmentation cells (Section 2.4).

The PF model must:

- maintain query predicates and associated usage information obtained from application sessions,
- provide a means of identifying primitive predicates and associated usage information,
- be able to handle horizontal, vertical and spatial fragmentations,
- be able to deal with compound predicates involving the AND, OR and NOT operators,

- be dynamically re-configurable to adapt to a changing application environment,
- provide operations to add new user predicates,
- provide an operation to delete predicates at both leaf and non-leaf nodes, and
- have provision to remove fragments that are not required.

3.3 Description of the PF Model

The PF model is implemented using a *PF tree*. A binary tree structure was chosen as the basis for the PF tree for the following reasons:

1. the binary tree is relatively easy to implement,
2. it has well-known operations (*eg. insert, delete*) defined on it (Aho and Ullman 1992),
3. the tree can be dynamically altered by adding or deleting nodes and therefore can adapt to a dynamic environment where fragmentation is changing, and
4. the hierarchical nature of a tree is well suited for accumulating information up or down the branches.

A PF tree may represent the fragmentation of an entity. A separate tree is defined for each entity being fragmented. The root node of a tree represents an entire entity set and each non-root node is associated with a predicate that defines a subset of the entity set. A primitive predicate is identified as the conjunction of predicates along the path from the root node to a leaf node identifying a unique fragment of the entity set. Therefore there are as many primitive predicates as leaf nodes. The leaf nodes are connected in a linked-list structure to facilitate processing of the primitive predicates (Figure 3.1).

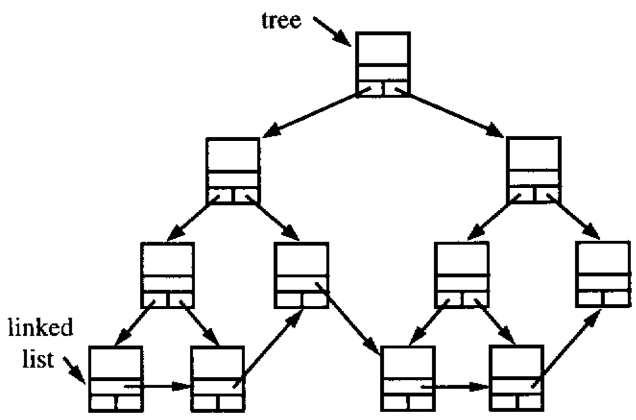


Figure 3.1 PF tree structure and linked list

The PF tree is constructed from predicates obtained from user applications. Compound predicate expressions are broken down into multiple simple predicate expressions so that they can easily be manipulated in the PF implementation.[3] Each simple predicate is then associated with a node and is inserted into the tree according to specified insertion rules (see Section 3.3.1).

It is possible to leave a compound predicate expression without breaking it down only if it is *never* broken down further or the terms comprising the predicate expression are *always* used together. For example, consider the following compound predicate expression that defines a rectangular geographic area of interest:

> *X >= 430000 AND X <= 440000 AND Y >= 6355000 AND Y <= 6365000*

Although this predicate could be broken down into four simple predicates, the X and Y coordinates are often used together. Further, in specifying the rectangular region, the upper and lower limits in both the X and Y directions are always specified together. In such a case the compound expression can be treated as a simple expression for the purposes of the PF tree implementation.

The PF tree structure can be illustrated by an example. Consider the following user predicate defined on an entity called *mines*:

> *mapsheet = 'Laverton' AND mineral = 'Au'* [1]

This compound predicate can be broken into two simple predicates as follows:

> *mapsheet = 'Laverton'*
> *mineral = 'Au'*

The PF tree corresponding to these two is shown in Figure 3.2. Section 3.3.1 explains how the tree is constructed from such predicates. Note that two nodes are defined for each predicate; one node associated with the predicate itself and represents the entity instances selected, and the other node associated with the converse of the predicate and represents the entity instances not selected. By including the *converse* nodes, all entity instances are accounted for in the PF tree as required by the fragmentation rules (Section 2.4.2). All non-leaf nodes must therefore have two children, one child being the converse of the other.

The converse nodes introduce more splitting and more nodes into the tree. It is possible to implement the PF tree without inserting the converse nodes. Essentially the nodes in the tree would then represent only predicates that were actually used in the applications. The children of a node could then be associated with two different and unrelated predicates. Also, the size of the tree could be reduced somewhat, although optimisation strategies do this

[3] Note that even though simple predicate expressions cannot be broken down further, the data that they define yet may be overlapping with that of other predicates (see Section 2.4.2).

anyway by removing redundancy and unwanted fragments (see Section 3.3.2). However, the linked list implementation would become more complex since non-leaf nodes (*ie.* nodes with only *one* child) must then be included to identify the primitive predicates. More importantly, the converse nodes are used to maintain application usage information, in particular the size of the (unaccessed) fragment, which is necessary to determine which fragments are (not) required (see Section 3.4.5). For these reasons it is better to explicitly place the converse node in the PF tree.

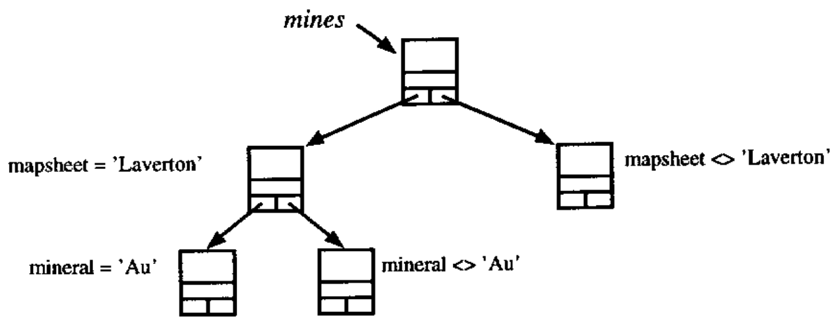


Figure 3.2 Example of a PF tree for *mines*

The predicates of parent-child nodes are joined by the AND operator. In the example of Figure 3.2 the three primitive predicates obtained for the entity *mines* are:

(mapsheet = 'Laverton') AND (mineral = 'Au')
(mapsheet = 'Laverton') AND (mineral <> 'Au')
(mapsheet <> 'Laverton')

The construction of the PF tree is dependent on the order in which the predicates are added to the tree. The result is a different fragmentation for data that is not accessed by the user applications or a further fragmentation to identify the data being accessed. This can be shown through a number of examples.

Suppose that the predicates in the example of [1] were given in reverse order as follows:

mineral = 'Au' AND mapsheet = 'Laverton' [2]

The primitive predicates obtained would be:

(mineral = 'Au') AND (mapsheet = 'Laverton')
(mineral = 'Au') AND (mapsheet <> 'Laverton')
(mineral <> 'Au')

Because of the AND operation, the order of the predicates in the primitive predicate expression does not affect the fragment being accessed. It is the other two fragments involving the converse nodes that have been affected. Because they are not accessed by the user query of [2] the effect of the revised fragmentation is minimal.

As another example, consider two independent user queries as follows:

mapsheet = 'Laverton' [3]

mineral = 'Au' [4]

The resulting PF tree would contain the following primitive predicates:

(mapsheet = 'Laverton') AND (mineral = 'Au')
(mapsheet = 'Laverton') AND (mineral <> 'Au')
(mapsheet <> 'Laverton') AND (mineral = 'Au')
(mapsheet <> 'Laverton') AND (mineral <> 'Au')

Now if the order of insertion for the predicates [3] and [4] were reversed, then the resulting PF tree would have the primitive predicates:

(mineral = 'Au') AND (mapsheet = 'Laverton')
(mineral = 'Au') AND (mapsheet <> 'Laverton')
(mineral <> 'Au') AND (mapsheet = 'Laverton')
(mineral <> 'Au') AND (mapsheet <> 'Laverton')

Although the predicates are placed differently within a PF tree, the primitive predicates and the fragments they define are the same.

A final example involves a predicate that may happen to already exist in the tree. Suppose that the following (independent) user query occurred after that of [1] (*ie.* the PF tree of Figure 3.2):

(mapsheet = 'Laverton')

A predicate associated with this query already exists and no modification is required to the PF tree. Now suppose that this query occurred after that of [2] where the predicates are inserted into the tree in reverse order. Now the primitive predicate "*mineral <> 'Au'*" will be replaced with two primitive predicates:

(mineral <> 'Au') AND (mapsheet = 'Laverton')
(mineral <> 'Au') AND (mapsheet <> 'Laverton')

Because of the order of insertion, one or more fragments may require splitting to accommodate the new predicate. If the fragments contain no data, the fragmentation will be removed in the pruning stages (Section 3.3.2). Note that this fragment splitting only occurs when a user predicate happens to equate with one or more existing primitive predicates.

The effects of the insertion order of predicates are further investigated in Chapter 5 where user queries are randomised and the resulting PF trees are examined. The effects are expected to be marginal since either only non-accessed data is affected or the existing primitive predicates happen to coincide with a new user query. To deal with this, an optimisation strategy would have to anticipate the best likely ordering based on previous (or future) queries. Such a strategy could be a topic of further research.

The primitive predicates collectively define all of the entity instances of the original entity *mines*. Furthermore the fragments defined by these primitive predicates are disjoint. The only exceptions to the "disjoint" rule are key fields and spatial primitives as explained in Sections 2.4.2.1 and 2.4.2.3. The original entity can be reconstructed by taking the union of all the fragments (defined by the disjunction of all the primitive predicates). The PF tree defines a correct fragmentation since all the fragmentation rules are adhered to.

The PF tree has similarities with both a *query execution tree* and a *fragmentation tree*. A query execution tree is an established method for describing operations on an entity set (Ceri and Pelagati 1984; Yu and Chang 1984; Ozsu and Valduriez 1991b; Adam and Gangopadhyay 1997). Both trees are similar in that the nodes represent a predicate and are related to other nodes through a set operator. The difference lies in that a query execution tree is used to represent a compound predicate expression based on multiple simple expressions for the purpose of query execution planning, while a PF tree is used to build compound predicate expressions (primitive predicates) from simple expressions (user predicates) to identify a fragmentation. A fragmentation tree defines how an entity set is partitioned based only on horizontal and vertical fragmentation (Ozsu and Valduriez 1991b). The nodes define the fragment and the operations relating nodes are horizontal and vertical fragmentation operations. The PF tree extends the concept of the fragmentation tree to encompass spatial data and provide a means of maintaining application usage properties.

3.3.1 PF tree construction

A PF tree is constructed from the predicates of user applications. The predicates are specified within selection operators that indicate how the data is to be selected. The PF model must use this selection information to construct the tree (by node insertion). Three selection operators are recognised by the PF model: *reselect, aselect* and *nselect*. See Section 2.5 for details on these operators. The use of these operators makes the PF tree simple to implement and maintain.

In order to understand how these operators are used by the PF model, the concept of an *active* node is introduced. Recall that these operators only execute on the *current selection* of data for a particular entity set and that the current selection is altered only with another selection operation on that same entity. An active node indicates that the data defined by a primitive predicate is contained in the *current selection* of data. Conversely, an *inactive* node indicates that this data is not found in the current selection. Only leaf nodes are considered active or inactive since all of the primitive predicates can be identified from them.

Figure 3.3 illustrates the effect of these operations on both a data entity and its associated PF tree. Active nodes are indicated with an asterisk. Note that the tree does not increase in size with the *nselect* operation whereas the *reselect* and *aselect* operations may result in tree growth. For the *reselect* operation, tree growth occurs only on nodes associated with the current selection whereas for the *aselect* operation tree growth occurs at nodes not associated with the current selection. Growth only occurs when a predicate needs to be inserted into the tree.

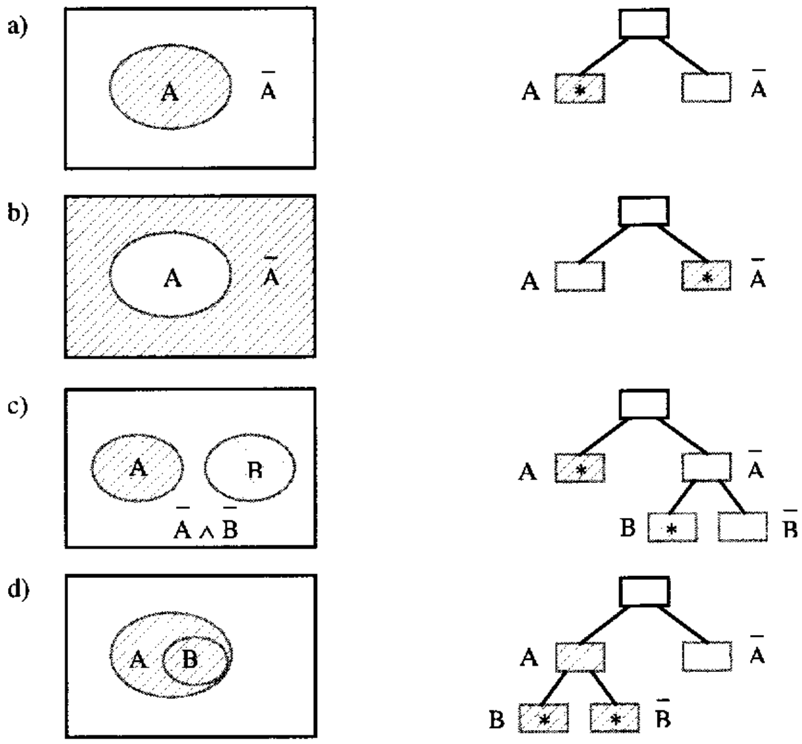


Figure 3.3 Effect of the selection operations when inserting predicates in a PF tree: a) the current selection on an entity set, b) nselect, c) aselect, and d) reselect.

Predicates are only inserted into the tree for active primitive predicates. For each primitive predicate, the predicate to be inserted must not already exist. If it does already exist, all that needs to be done is to update the usage information in the predicate nodes. To determine if a predicate already exists within a primitive predicate, all the predicates comprising the primitive predicate must be searched. The predicate exists if either an identical or an equivalent predicate is found. An *equivalent* predicate is one that doesn't look the same (usually because different or inverse operators are used) but refers to the same data. The following examples provide predicates that are equivalent:

mapsheet = 'Laverton'	is equivalent to	*NOT (mapsheet <> 'Laverton')*
production > 45	is equivalent to	*NOT (production <= 45)*

Note that it is not possible to identify *all* equivalent predicates from simply analysing the predicate expressions. Two completely different queries can still refer to identical data. For example the predicates "*production > 45*" and "*status = 'Developing*" may both refer to the same data. This information is only possible to determine by querying the data and comparing outcomes. This is done in the pruning stage explained in Section 3.3.2 and implemented in Section 4.4.3.

When compound predicates are broken down into simple predicates, the appropriate selection operations must be used so that the predicates are properly inserted into the tree. Recall Section 2.5 which describes the relationship between the three selection operations used by the PF tree and the relational SQL operators. The NOT operator corresponds directly with the *nselect* operator. The AND operator in a compound expression is equivalent to two *reselect* operations and the OR operator corresponds with one *reselect* and one *aselect* operation.

For example, the compound predicate *(mapsheet = 'Laverton') AND (mineral = 'Au')* would be translated to the following operations with simple predicates:

reselect mapsheet = 'Laverton'
reselect mineral = 'Au'

and the predicate *(mapsheet = 'Laverton') OR (mineral = 'Au')* would be translated to:

reselect mapsheet = 'Laverton'
aselect mineral = 'Au'

The cells that are identified in a geographic data fragmentation are potentially a combination of horizontal, vertical and spatial fragmentations and therefore the PF model must allow for all possibilities. Horizontal and spatial fragmentation are catered for in the PF model through the use of the selection operators. A selection identifies entity instances that include both the rows of an attribute table and the spatial entities from the spatial data table.

3.3.2 Predicate pruning

Not all primitive predicates produced by a PF tree identify fragments that will be used in a data distribution. Some primitive predicates may have to be altered or removed by deleting predicates. This process is referred to as *pruning*. Pruning may be necessary to discard predicates that refer to fragments that are either redundant, too small, empty, infrequently accessed, outdated or are allocated to the same site.

An important advantage of pruning is that the tree is not allowed to expand unchecked. Tree growth is potentially explosive as the number of predicates being inserted increases. A large tree will result in slower processing time. Pruning is important to remove predicates that are not necessary to be maintained in the tree and as a means of tree optimisation.

A PF tree may require pruning in the following situations:

> **Predicate expression redundancies and conflicts**. If a predicate is equivalent (Section 3.3.1) to another predicate in a primitive predicate, then it is *redundant* and does not need to be inserted in the PF tree for that primitive predicate. A predicate *conflicts* with another predicate when it is impossible for an entity instance to be identified by both predicates. Often it can be obvious from examining the predicate expressions that a conflict occurs. For example, the predicate *mapsheet = 'Laverton'* is in conflict with *mapsheet = 'Augusta'* since it is impossible for a map sheet to have two names. As another example, the predicate *production > 50* obviously conflicts with the predicate *production <= 25*.
>
> Redundancies and conflicts, which can be identified from the predicate expressions, can be resolved, using character string manipulation techniques, before the relevant predicates are inserted into the tree. However, redundancies and conflicts can also occur where the corresponding predicate expressions give no indication of this. As explained in Section 3.3.1, this can be achieved by comparing query outcomes.
>
> **Primitive predicates that result in small or empty fragments**. Even when two predicates do not conflict, it is possible that a valid primitive predicate defines an empty fragment. Some fragments may also contain such a small number of entity instances that the cost of maintaining the fragmentation exceeds the benefits. To determine the size of a fragment, a query involving the primitive predicate must be executed on the entity set. Any fragmentation that results in empty fragments or in fragments with fewer instances than a given minimum *threshhold* can be removed from the PF tree. The threshhold is related to the fragment size and frequency of

accesses and is the point at which the cost of maintaining the fragment becomes greater than the additional cost involved had the fragmentation not occurred. The calculation of this threshhold value is detailed in Section 4.3.

Primitive predicates that are infrequently accessed. Certain fragments may be accessed very infrequently so that the cost of maintaining such a fragmentation exceeds the benefits. Of course a fragmentation may occur simply to isolate the frequently accessed data from the infrequently accessed data. Therefore both the predicate and its converse predicate in the PF tree must be considered when determining the frequency of access. If both are infrequently accessed, then the fragmentation is not beneficial and the leaf node of the primitive predicate can be deleted. The threshhold value referred to above relates the fragment size to the frequency of accesses and can be used to determine when *infrequent access* occurs.

Predicates that are not recently accessed. As more new user predicates are obtained from the application logs, the PF tree will continue to grow. However, after some time, certain predicates within each primitive predicate may not be referred to anymore in the selection operations of current or most recent applications and hence are not *relevant* to specifying the primitive predicate. These predicates are often not the leaf nodes in the PF tree since they were previously obtained from logs of earlier applications. Both the predicate and its converse must be considered since the fragmentation may be valid if the converse predicate has been recently accessed but the predicate itself has not. When both the predicate and its converse have not recently been accessed, they can be removed from the tree, thereby simplifying the affected primitive predicate clauses and reducing the tree size.

Primitive predicates that are allocated to the same site. When the primitive predicates have been allocated to computer network sites in the data distribution process, it is possible that two primitive predicates are allocated to the same site. Because no inter-site transfer is required, it is assumed that such a fragmentation is not necessary and can be eliminated if the leaf nodes of the affected primitive predicates are siblings in a PF tree. Pruning can be carried out recursively until no more sibling nodes are allocated to the same site.

The predicates to be pruned are identified in a fragmentation filter process (Figure 3.4). The result is that some primitive predicates will merge to form new primitive predicates. Some application information is required for this process, in particular the fragment size, the frequency of usage for the fragment, and the time in which the fragment was last accessed. This information is used in executing the pruning methods described above.

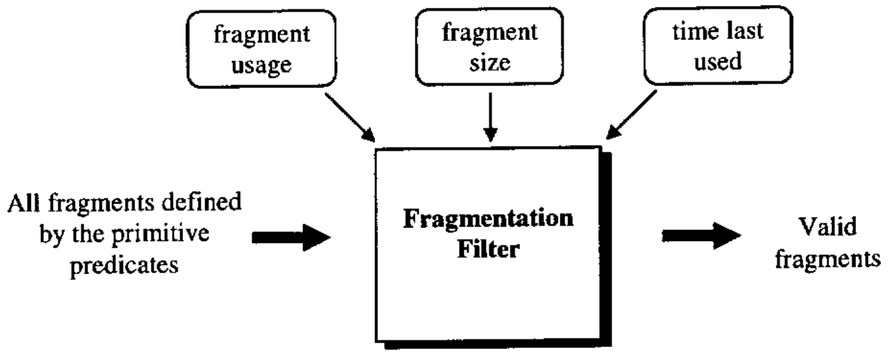


Figure 3.4 Fragmentation filtering process for identifying valid fragments

To implement pruning a *delete* operation is required for the PF tree. The delete operation executes on a particular predicate that is to be removed from the tree. The pruning of a primitive predicate does not mean that every predicate which comprises the primitive predicate needs to be removed. Rather, only the leaf node should be removed since the remaining predicates may still be used in other primitive predicates. The details of the delete operation are further discussed in Section 3.4.4 and some examples of pruning are provided in Section 3.4.5.

3.3.3 PF model extension

An extension to the PF model incorporates vertical fragmentation where the attributes of an entity are grouped into different fragments. In addition to allowing a tree node to refer to a predicate, it can also refer to a set of attributes that identify the vertical fragment. The addition of a vertical fragment to the tree results in two new nodes being inserted under an active leaf node. One node contains the attributes identified from a vertical fragment; the other node contains the remaining attributes. The structure of the rest of the tree and all other operations remain unchanged. A primitive predicate may now refer to both a conjunction of predicates (*ie*. a spatial/horizontal fragment) and a set of attributes (*ie*. a vertical fragment). Because the *project* operator is not adequately catered for within ArcInfo, this extension was not tested in this research.

Figure 3.5 uses the example from Figure 3.2 to insert a vertical fragment after the predicate *mapsheet = 'Laverton'* (which was then active - it is assumed that no other predicates have yet been inserted). The fragment was identified from a user application and refers to the attributes *mine_id* and *mineral*.

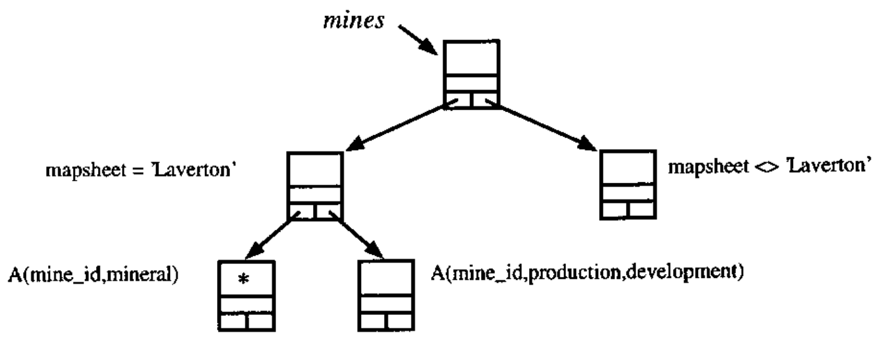


Figure 3.5 PF tree showing vertical fragmentation of attribute *mineral* from other attributes of entity *mines*

After the insertion, the node referring to this fragment becomes active. Note that the key field (*mine_id*) is replicated in the two new vertical fragments. Subsequent tree insertions, whether due to selection operations or vertical fragments, can continue to be made as usual.

3.4 Implementation of the PF Linked Tree

The PF model is implemented in the C programming language and uses dynamic storage to maintain the PF trees. One tree is generated for each entity used by the applications. For each tree, a tree pointer that identifies the root node and a list pointer that identifies the left-most leaf node is maintained. Each node in the tree is associated with a predicate and contains usage information about either the predicate or the primitive predicate with which it is associated. The general structure of a node is indicated in Figure 3.6.

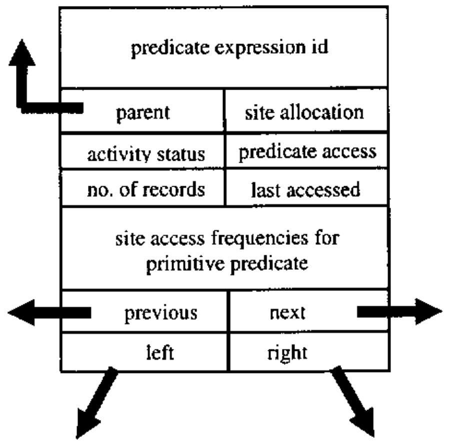


Figure 3.6 Structure of a PF tree node

A number of pointer fields are required to facilitate the implementation of the linked tree. The *left, right* and *parent* fields are pointers used for connecting the nodes in a binary tree structure. The *previous* and *next* fields are pointers used for maintaining a linked list of the leaf nodes. The linked list facilitates sequential access to the primitive predicates. The two latter fields are not required for non-leaf nodes.

The information associated with the predicate in each node consists of the following:

- predicate id which is used to identify the predicate from a separate predicate list,
- frequency of access for the predicate (ie. where predicate was explicitly referred to in selection), and
- time at which the predicate was last accessed.

The predicate access frequency information relates to predicates that are explicitly referred to in a selection operation. Note that the information at the root node is associated with the entire entity set (ie. selections containing *all* the entity instances). This usage information along with the time indicating when the predicate was last accessed is used in the pruning process to determine whether or not a predicate should be retained in a PF tree.

The following information is associated with each primitive predicate:

- frequency of access for the primitive predicate at each computer network site,
- size of fragment defined by the primitive predicate,
- site to which the fragment is allocated, and
- activity status

Apart from the access frequencies, this information is contained in the leaf node for each primitive predicate. As the tree grows through insertions of predicates, the appropriate information is transferred to the new leaf nodes. The size of a fragment is obtained by actually executing a selection operation, which uses the appropriate primitive predicate, on the entity set and observing how many entity instances are selected. The site at which the fragment is placed is determined in the site allocation process that is further described in Section 4.2.3.

The access frequency for each primitive predicate and for each site is maintained among all the nodes comprising the primitive predicate. The *access frequency* simply refers to the number of times a particular fragment of data (a primitive predicate in this case) is accessed. If only one primitive predicate is accessed, its usage is updated in the leaf node associated with that primitive predicate. If multiple primitive predicates are accessed together, then the access frequencies are updated in the node which is highest in the tree and whose associated primitive predicates are all active (*ie.* were all accessed together). Therefore the total number of accesses for a primitive predicate are obtained by summing the access frequencies in each of the nodes comprising the primitive predicate.

As an example the PF tree in Figure 3.7 contains the access frequencies for the primitive predicates for one particular site. In this example 42 accesses were made to all the entity instances in the entity set (*ie.* where all primitive predicates were concurrently active) and the fragmentation defined by "*mapsheet = 'Laverton*'" was accessed 25 times. Because the entity instances of this fragment were accessed as a fragment as well as together with all the entity instances, the total number of accesses to the fragment "*mapsheet = 'Laverton*'" is 67 (25+42). Similarly, the total number of accesses to the fragment "*(mapsheet = 'Laverton') AND (mineral = 'Au')*" is 135 (68+25+42).

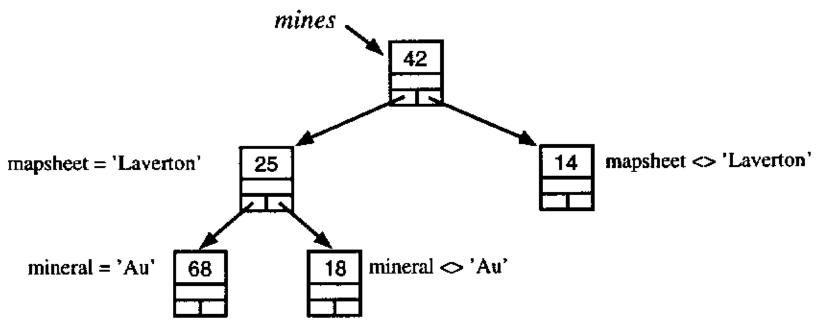


Figure 3.7 Access frequencies for primitive predicates

As predicates are inserted into and deleted from the PF tree, it is important to maintain the usage information. For node insertions, the access frequency information is maintained in the existing nodes, but for deletion operations this information is transferred to the parents of deleted nodes. The assumption is that had the fragmentation not occurred, the fragment identified by the parent would have been accessed instead. The insertion and deletion operations are further detailed in the following sections.

3.4.1 Reselect operation

Given that all or some (or even no) entity instances are contained in a *current selection*, the *reselect* operation defines a selection on the current selection. Therefore, only the *active* primitive predicates in the tree need to be traversed. For each active node, a *reselect* operation will result in two new child nodes, one defining the new selection, and the other defining the converse. The node defining the new selection will be tagged as *active* and the other as *inactive*.

Consider the example in Figure 3.8. Assume that there was only one active primitive predicate for the entity *mines*, namely, "*(mapsheet = 'Laverton') & (mineral = 'Au')*". A reselection operation such as:

reselect mines production > 10

will only affect the one node that is active. Two child nodes result, the left one with the predicate "*production > 10*" and the right one with the inverse predicate "*production <= 10*".

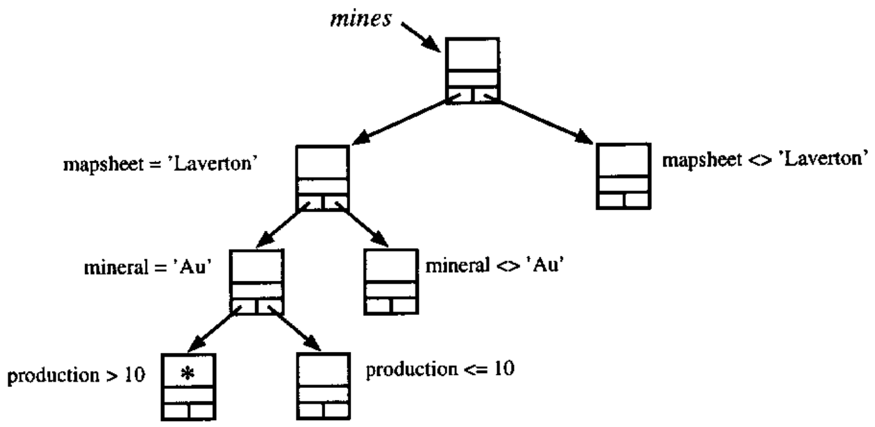


Figure 3.8 Reselection operation with predicate "*productivity > 10*"

Therefore two new primitive predicates, the one active and the other inactive, replace the former active primitive predicate. For this example the two new primitive predicates are:

(mapsheet = 'Laverton') & (mineral = 'Au') & (productivity > 10)
(mapsheet = 'Laverton') & (mineral = 'Au') & (productivity <= 10)

These two primitive predicates are disjoint. They are also complete since they include all of the data instances of the primitive predicate just replaced.

The procedure to insert into a PF tree based on a reselection operator is outlined in the following pseudo-code:

```
procedure insert_resel(list,pred)
while list ≠ NULL
        if list->active
                if NOT conflict(pred,list) or redundant(pred,list)
                        insert_node(list->left, pred)
                        insert_node(list->right, NOT pred)
                endif
                update_usage(pred,list)
        endif
        list = list->next
```

```
endwhile
endprocedure
```

The variable *list* refers to the linked list connecting the leaf nodes of the PF tree while *pred* refers to the new predicate to be inserted into the tree. For each primitive predicate, the new predicate must be checked to determine if any conflicts or redundancies are evident among the predicates comprising the primitive predicate. This is achieved by using simple string manipulation techniques to identify the operators and operands, and checking for possible equivalence (Section 3.3.1) or a conflict (Section 3.3.2). If a conflict or redundancy occurs, then the usage information for the existing predicate and primitive predicate must be updated and the primitive predicate must be activated. Otherwise, two new nodes are inserted and the usage statistics initialised for each. Initialisation involves setting the predicate access and primitive predicate access (for the appropriate site) counts to 1 and inserting the time of access. The size of the fragment does not have to be obtained until later when the fragmentation filtering process is executed. Of course, the site at which the fragment is placed cannot be determined until the allocation process is conducted.

3.4.2 Aselect operation

The "add-to-selection" or *aselect* operator adds, to the current selection, instances from the set of entities that are not selected and which are defined by a given predicate expression. If no expression is given in the *aselect* command then all the remaining unselected instances of an entity are added to the current selection or, in other words, *all* instances are selected. Therefore, when updating a PF tree based on an *aselect* operation, only the inactive nodes need to be considered. If all the entity instances are to be selected, then no further primitive predicates need to be formed and all need to be activated.

If, however, a predicate expression is supplied with the *aselect* operator, then the predicate will specify a subset of the unselected instances to be added to the current selection. Therefore, all inactive primitive predicates must be replaced with two new primitive predicates in a manner similar to that for the *reselect* operation. The node(s) associated with the new predicate is tagged as active while the node(s) containing the inverse predicate is tagged as inactive. In effect, a *reselect* operation takes place for each inactive primitive predicate.

As an example, consider the PF tree of Figure 3.9 and assume that only the node containing "*production* > *10*" is active. Then suppose that the following *aselect* operation occurs:

```
aselect mines
```

which indicates that all instances of the entity *mines* are to be selected. All that needs to be done to the PF tree is to activate all inactive leaf nodes as shown in Figure 3.9.

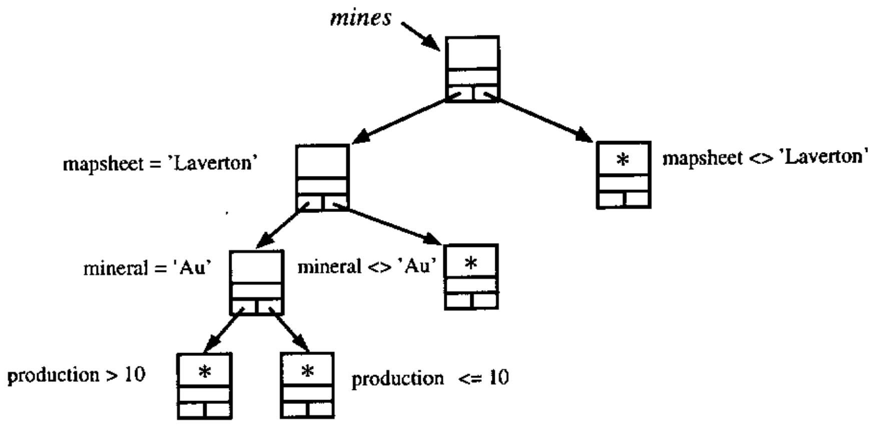


Figure 3.9 *Aselect* operation with no predicate expression

If the *aselect* operation includes a predicate expression as in the following:

aselect mines type = 'Developed'

then the insertion of a new predicate is applied to each inactive node. Assume, once again, that only the node containing "*production > 10*" is initially active, the result is that three inactive primitive predicates are replaced with three active and three inactive primitive predicates as shown in Figure 3.10. Note that not all these primitive predicates may define valid fragments and could therefore be pruned (see sections 3.3.2 and 3.4.4) resulting in a simpler tree.

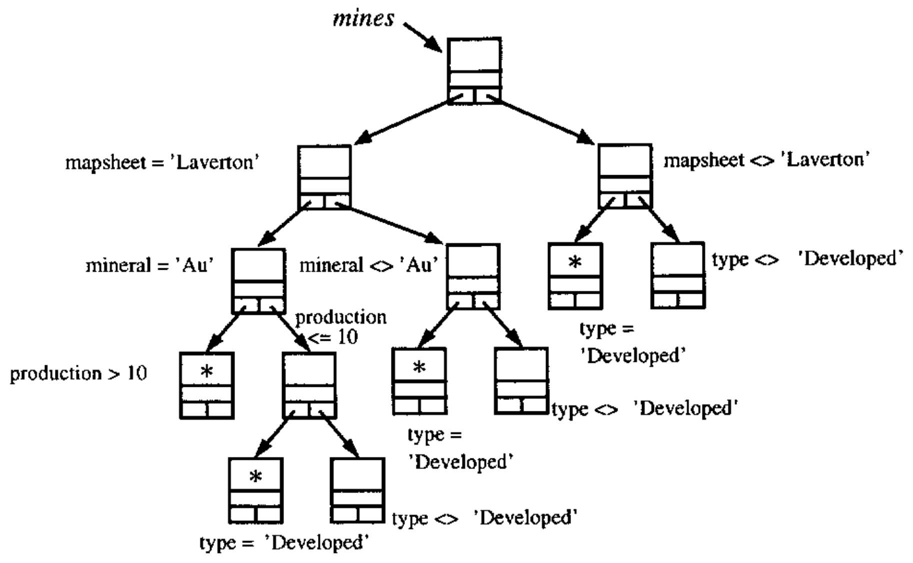


Figure 3.10 *Aselect* operation with predicate "*type= 'Developed'*"

In a more general logical expression tree the appearance of an *aselect* operation would mean the inclusion of an OR term in each predicate clause resulting in a more complex predicate expression. By generating children from only inactive parents, the clause remains a simple conjunctive clause and the insertion process is simplified.

The procedure for implementing a PF tree insertion based on the *aselect* operator is as follows:

```
procedure insert_asel(list,pred)
while list ≠ NULL
          if NOT list->active
          if pred is empty expression
                    if NOT conflict(pred,list) or duplicate(pred,list)
                              insert_node(list->left, pred)
                              insert_node(list->right, NOT pred)
                    endif
          else
                    list->active = TRUE
                    endif
                    update_usage(pred,list)
          endif
          list = list->next
endwhile
endprocedure
```

As with the reselection operator, before a predicate can be inserted, the primitive predicate must be checked to determine if any conflicts or redundancies occur. If not, the predicate is inserted. The usage information is updated for the predicate and associated primitive predicate (as for the *reselect* operation) and the primitive predicate is activated.

3.4.3 Nselect operation

The *nselect* operator negates the selected instances of the entities; that is, all the selected instances become unselected and the previously unselected instances now form the current selection. For the PF tree, this simply requires activating all the inactive leaf nodes and deactivating all the active nodes. The result of applying the *nselect* operation to the PF tree of Figure 3.10 is the tree as shown in Figure 3.11.

In a more general logical expression tree the appearance of an *nselect* operation would mean the inclusion of a NOT operator which would need to be applied over the current predicate clause resulting in a more complex predicate expression. By simply switching the active status of the leaves the clause remains a simple conjunctive.

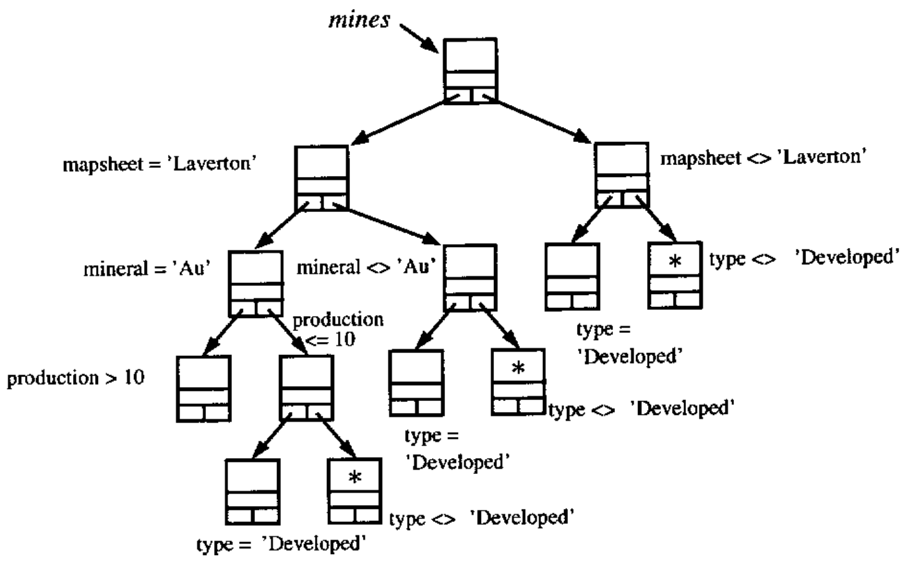


Figure 3.11 The effect of the *nselect* operation on the entity *mines*

The procedure for implementing the *nselect* operation is shown using pseudo code as follows:

```
procedure insert_nsel(list)
while list ≠ NULL
    list->active = NOT list->active
    update_usage(list)
    list = list->next
endwhile
endprocedure
```

The *nselect* operation does not involve the insertion of new nodes and hence does not affect tree growth.

3.4.4 Delete operation

The deletion of a node in the PF tree is the removal of a predicate from one or more primitive predicates. Because the inverse predicate is inserted as a sibling node when a predicate is inserted, this sibling node must also be removed when the predicate is deleted. A deletion implies that the fragmentation between data instances defined by the predicate and its inverse is no longer necessary. For example, if the predicate "*type* = 'Developed'" is being removed, then the inverse predicate "*type* <> 'Developed'" found in the sibling node also needs to be removed.

The deletion operation involves much more than simply removing two nodes from the tree. The subtrees of any non-leaf nodes must be maintained since they define further predicate expressions within the primitive predicates. In addition, usage information for the remaining predicates and primitive predicates must be retained.

If the node and its sibling node being removed are both leaf nodes, the deletion operation simply requires adjusting the tree and linked list pointers (Figure 3.12). If both nodes being removed are active, then the parent node, which is now the leaf node, will become active. Similarly if both nodes are inactive, then the parent node should be inactive.

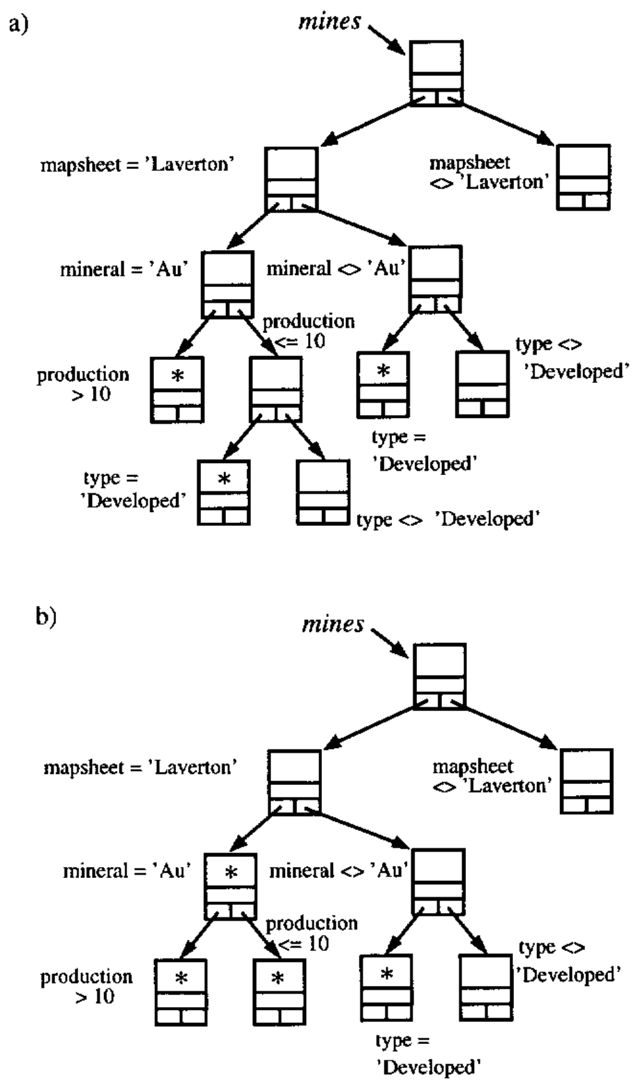


Figure 3.12 Deletion of *one* instance of predicate "*type = 'Developed'*" a) before deletion, and b) after removing two leaf nodes

If however only one of the nodes is active, then it is not directly obvious as to what the status of the parent node should be. If one child node is active, this implies that a subset (defined by the predicate of the active child) of the data instances defined by the parent is being accessed while the remainder is not. If this predicate is no longer necessary, then it can be assumed that the parent node will retain its (former) child's active status, since it is now associated with the data instances previously associated with its (former) child. Of course, the entity instances associated with the (former) inactive child are also now included with the parent node. The effect of the *delete* operation is to deal with the parent node as if no further fragmentation had occurred. The result of a deletion is that one primitive predicate is removed and the other one that is affected has one predicate removed. Given the example in Figure 3.12, the primitive predicates that are affected by the delete operation (of only the left most instance of the predicate) are as follows:

> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production > 10)*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production <= 10) & (type = 'Developed')*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production <= 10) & (type <> 'Developed')*

After the delete operation is applied, the primitive predicates that exist are:

> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production > 10)*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production <= 10)*

Now consider the situation where one of the pair of nodes being removed has child nodes. These child nodes now become children to the parent of the nodes being removed. Suppose for example, in Figure 3.13, that the node "*production > 10*" is to be deleted. The sibling node "*production <= 10*", which will also be removed, has two child nodes. These child nodes will now be linked to the parent "*mineral = 'Au'*".

The effect is that one primitive predicate is dropped whereas the other primitive predicates affected have one predicate expression removed. The two modified primitive predicates now appear as follows:

> *(mapsheet = 'Laverton') & (mineral = 'Au') & (type = 'Developed')*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (type <> 'Developed')*

During this process, the number of leaf nodes does not change. The only difference is that the associated primitive predicates have one less predicate clause. Therefore that *active* status of the associated primitive predicates does not change. Leaf nodes that were active remain active, and inactive ones remain inactive. The status of the leaf node that was removed is not relevant any more and therefore need not be retained, since the associated predicate clause has now been removed.

The entity instances, and therefore also the associated usage information, defined by the removed leaf node (*ie*. the primitive predicate that was dropped) are now associated with the new primitive predicates. Therefore it is assumed that the access counts of the removed primitive predicate are added to *each* new primitive predicate, in addition to the usage information they already contain. The exception is when the deletion is due to predicates that have not recently been accessed.[4] In such a case the usage information is outdated and can be discarded by not revising the usage information for the new resultant primitive predicates.

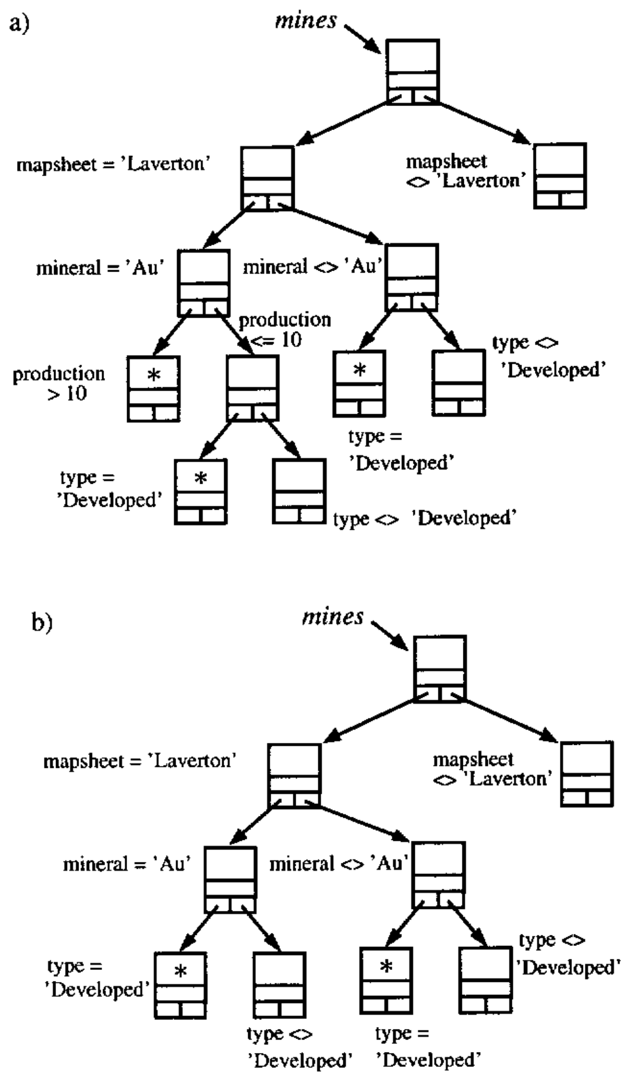


Figure 3.13 Deletion of predicate "*production > 10*" a) before deletion, and b) after removing a leaf and a non-leaf node

Finally consider the situation where the pair of nodes to be removed both have child nodes. Both subtrees of the pair of nodes define further reselections that must be maintained. The subtree from one of the child nodes being deleted can now become the subtree of the parent

[4] Note that this is one of the pruning methods discussed in Section 3.3.2.

node. To retain the predicate clauses associated with the other child being removed, the subtree must be inserted under each of the leaf nodes of the first subtree.

For example, suppose that the node containing "*mineral* = *'Au'*" were to be deleted in Figure 3.14. This node, as well as its sibling node containing the inverse predicate, have child nodes which themselves define subtrees. Suppose that the two subtrees of the node "*mineral* = *'Au'*" are placed under its parent "*mapsheet* = *'Laverton'*". Then the subtrees of the sibling node "*mineral* <> *'Au'*" can be placed under each leaf node of those subtrees. The process is illustrated in Figure 3.14 where the subtrees are identified with different shading patterns to identify their placement in the tree. Note that for the purposes of illustrating this deletion, this example does not consider redundant and conflicting predicates (which are evident and explained below).

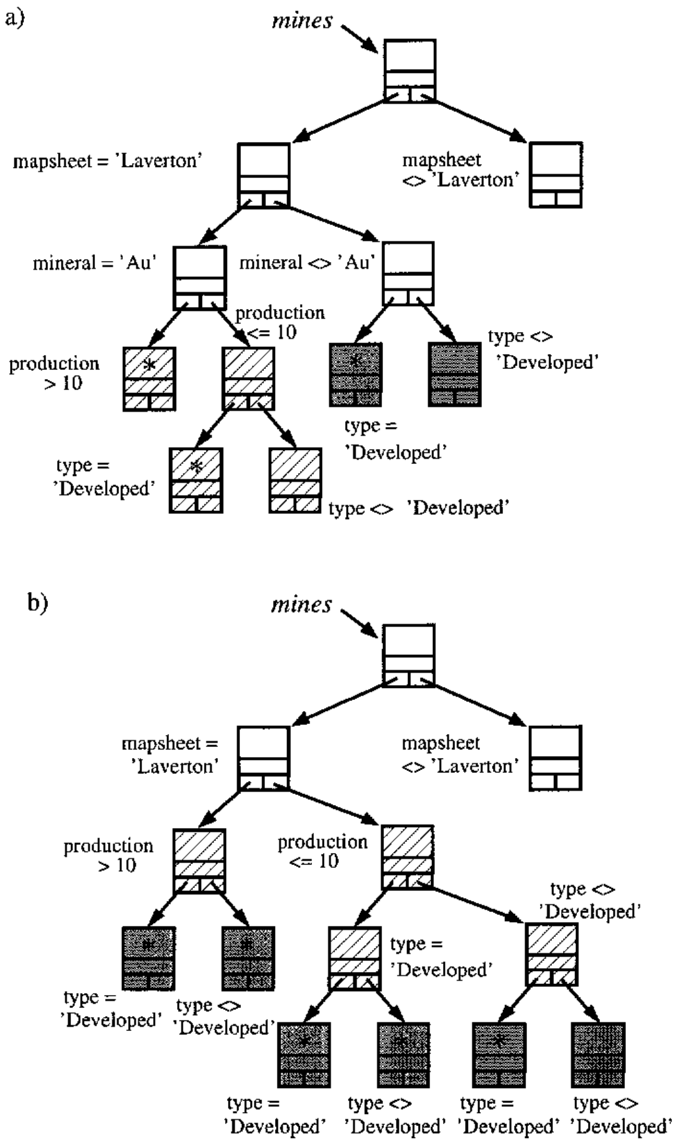


Figure 3.14 Deletion of predicate "*mineral* = *'Au'*" a) before deletion, and b) after removing two non-leaf nodes

All leaf nodes which were also leaf nodes before the deletion operation took place will remain active or inactive as they were previously. The exception is where one of its ancestors was previously an active leaf node. In such a case the leaf node represents a fragment of an already selected fragment of data and therefore must be active to reflect the fact that it already belongs to the current selection.

For example, in Figure 3.14 the primitive predicate "*mineral* = *'Au' and production* > *10*" was selected before the deletion operation. By removing the predicate "*mineral* = *'Au'*", the redefined primitive predicate "*production* > *10*" remains selected. Thus any further fragmentation caused by adding predicates (due to the deletion operation) should reflect the fact that this whole fragment is already selected. All leaf nodes under this subtree should therefore have an active status. As indicated by the example, both "*type* = 'Developed'" and "*type* <> 'Developed'" nodes are active.

The primitive predicates affected by the deletion in this example are:

> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production > 10)*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production <= 10) & (type = 'Developed')*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (production <= 10) & (type <> 'Developed')*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (type = 'Developed')*
> *(mapsheet = 'Laverton') & (mineral = 'Au') & (type <> 'Developed')*

The result of the deletion are the following primitive predicates:

> *(mapsheet = 'Laverton') & (production > 10) & (type = 'Developed')*
> *(mapsheet = 'Laverton') & (production > 10) & (type <> 'Developed')*
> *(mapsheet = 'Laverton') & (production <= 10) & (type = 'Developed') & (type = 'Developed)*
> *(mapsheet = 'Laverton') & (production <= 10) & (type = 'Developed') & (type <> 'Developed)*
> *(mapsheet = 'Laverton') & (production <= 10) & (type <> 'Developed') & (type = 'Developed)*
> *(mapsheet = 'Laverton') & (production <= 10) & (type <> 'Developed') & (type <> 'Developed)*

In this example, the number of primitive predicates has increased. This number is very dependent on the size of the subtrees of the pair of nodes being removed.

The deletion is not yet necessarily complete. Some pruning may be required and would lead to more nodes being removed and hence a decrease in the number of resulting primitive predicates. Notice that in this example the resulting primitive predicates contain redundant and conflicting predicates. Leaf nodes containing such redundancies or conflicts are pruned resulting in the tree illustrated in Figure 3.15. The primitive predicates now are:

(mapsheet = 'Laverton') & (production > 10) & (type = 'Developed')
(mapsheet = 'Laverton') & (production > 10) & (type <> 'Developed')
(mapsheet = 'Laverton') & (production <= 10) & (type = 'Developed')
(mapsheet = 'Laverton') & (production <= 10) & (type <> 'Developed')

Obviously there is a decrease in both the number of primitive predicates and corresponding size of the PF tree.

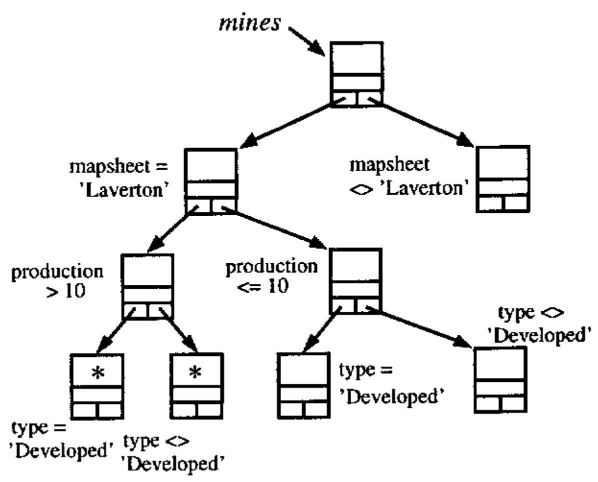


Figure 3.15 Pruning of predicates for a further reduction in tree size

Further pruning methods, as detailed in Section 3.3.2, may result in even further reductions in the number of primitive predicates and corresponding tree size.

The *delete* operation is shown in pseudo code as follows:

```
procedure delete_pred(list,node)
p = node (which must be a left sibling)
q = p->parent->right
if both p and q have no children
    make p->parent a leaf node - modify tree and list structure
    p->parent->active = p.active
elseif only p has no children
    connect subtree of q onto p->parent - modify tree and list structure
    s = leftmost leaf node under p->parent
    for each s
        s->usage = s-> usage + p->usage
        s = s->next
    endfor
```

```
elseif only q has no children
          connect subtree of p onto p->parent - modify tree and list structure
          s = leftmost leaf node under p->parent
          for each s
                    s->usage = s-> usage + q->usage
                    s = s->next
          endfor
else /* both p and q have children */
          connect subtree of p onto p->parent - modify tree and list structure
          for each node r under p
                    r->usage = r->usage + q->usage
          endfor
          r = leftmost leaf node under p
          for each r
                    insert subtree of q under r - modify tree and list structure
                    if r->active
                              s = leftmost leaf node under r
                              for each s
                                        s->active = TRUE
                                        s = s->next
                              endfor
                    endif
                    for each node s under r
                              s->usage = s->usage + r->usage
                    endfor
                    r->usage = r->usage + q->usage
                    r = r->next
          endfor
endif
free(p,q)
endprocedure
```

Note that when both of the nodes being removed have subtrees, the nodes of one subtree are inserted under each leaf of the other subtree. In this process, the normal pruning methods of checking for predicate redundancies and conflicts must take place. Note also that if the leaf node r under which the subtree is being inserted has an active status, then all the new leaves will also be activated.

The usage information for the primitive predicates must be maintained in the deletion operation. For the pair of predicates being removed, their individual usage statistics are no longer necessary. Therefore the access frequency of the predicate and time of last access can be deleted. However the primitive predicate usage must be retained. This is achieved by adding the usage counts of each of the nodes being removed to the parent node. The assumption is that the parent fragment would have been accessed in place of the two fragments, had the fragmentation not occurred.

3.4.5 Using the PF tree operations

The use of the PF tree operations can be illustrated with a simple example. Consider the following selection operations and the number of associated accesses obtained from three application user sessions for the entity *mines*:

User Session 1	User Session 2	User Session 3
reselect mapsheet = "Laverton"	reselect mapsheet = "Laverton"	reselect mapsheet = "Albany"
12 accesses	12 accesses	38 accesses
reselect mineral = "Au"	reselect mineral = "Au"	nselect
22 accesses	22 accesses	10 accesses
aselect status = "Operating"	reselect production > 20	
18 accesses	45 accesses	

The number of accesses to each selection of entity instances is shown following each selection operation. Assume that the PF tree for *mines* is initially empty (*ie*. only the root node exists) and that the user sessions are processed in the order: User Session 1, User Session 2, User Session 3. Also assume that the predicates are inserted in the PF tree in the order in which the selection operations appear in the user session. Note that all nodes are active at the start of processing a new user session since we assume that all entity instances are contained in the current selection.

The result of processing User Session 1 is the PF tree illustrated in Figure 3.16. Five primitive predicates are created of which three are active ("*" = active; otherwise it is inactive).

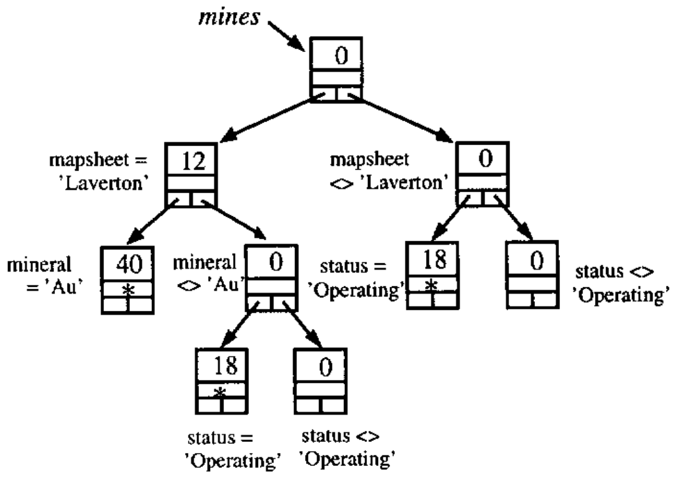


Figure 3.16 PF tree generated as a result of User Session 1

In addition to the activity status shown at the leaf nodes, each node shows the associated primitive predicate access frequency for a particular site. Note how the access count associated with the predicate "*mineral* = *'Au*" reflects the accesses which occurred after both the associated *reselect* operation and the *aselect* operation (ie. 22+18).

Because the first two reselection operations in User Session 2 are identical to those in User Session 1, the effect is simply to update the usage information and activate the one primitive predicate. The third reselection involving a new predicate causes an insertion into the tree. The result is six primitive predicates of which only one is active (see Figure 3.17).

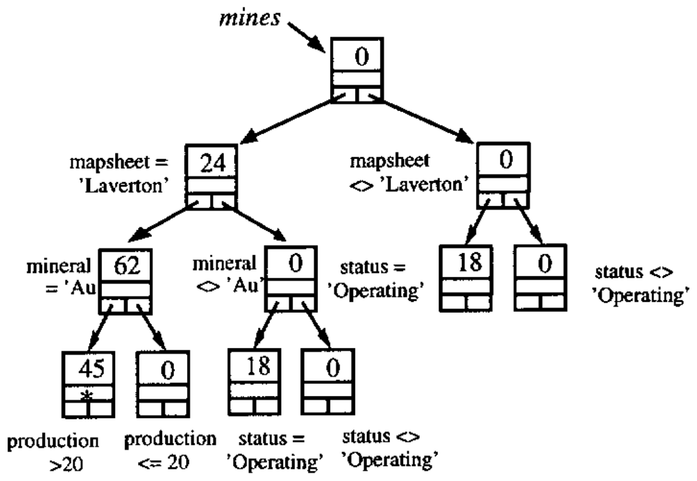


Figure 3.17 PF tree after processing User Session 2

The *reselect* operation in User Session 3 involves the *mapsheet* attribute but with a different value. The existing predicate "*mapsheet* = *'Laverton*" is in conflict with the new predicate and therefore its entire subtree can be disregarded in the insertion process, despite the fact that all primitive predicates are active. Because no predicates in the subtree "*mapsheet* <> *'Laverton*" are in conflict, the new predicates can be inserted at the corresponding leaf nodes. The result is eight primitive predicates of which two are active. The *nselect* operation in User Session 3 then causes the activity status of each leaf node to be inversed resulting in six active primitive predicates as shown in Figure 3.18.

Note how the primitive predicate usage information has been updated. The two active primitive predicates generated from the *reselect* operation have identical usage counts as would be expected. After the *nselect* operation, the other six primitive predicates are active and their usage counts are updated. Note that the usage count was updated for the predicate "*mapsheet* = *'Laverton'*" rather than at each leaf nodes, since each of the corresponding primitive predicates is active.

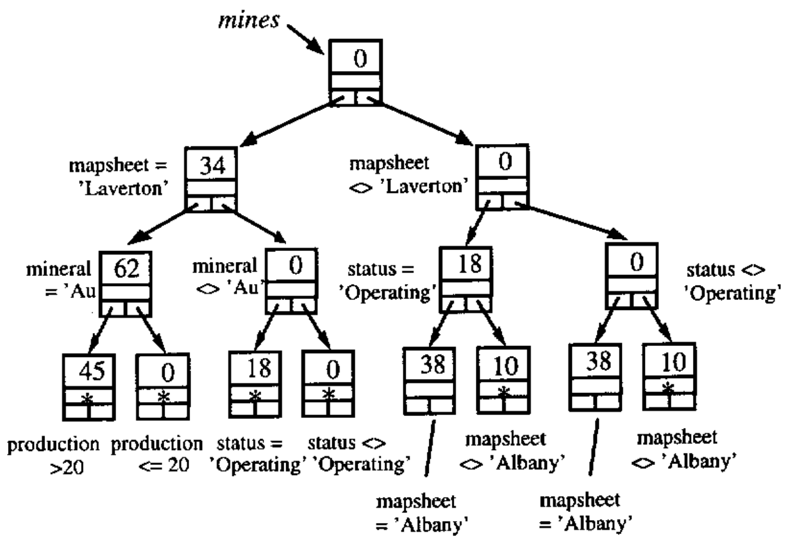


Figure 3.18 PF tree after processing User Session 3

The tree is not yet complete since some primitive predicates may define fragments that are not valid. Pruning operations may be necessary to filter out fragments according to the various pruning techniques described in Section 3.3.2. Consider an example where pruning is based on small and infrequently accessed primitive predicates. Assume that the size of the fragments defined by the six primitive predicates is determined to be 130, 0, 1, 25, 0, 572, 13 and 2340 entity instances[5], respectively from left to right. The total number of entities accessed[6] for each primitive predicate is therefore 5850, 0, 18, 0, 0, 5720, 494 and 23400, respectively. Assume also that for each of these primitive predicates the threshold value is calculated as equivalent to accessing 20 entity instances.

Both the predicate and its converse must be considered in determining whether the fragmentation is valid or not. Therefore any pair (predicate and its converse) of fragments where the combined number of entities accessed is smaller than the threshhold value must be deleted. The resulting tree is shown in Figure 3.19. Note that the number of primitive predicate accesses have been retained by the parents of the deleted nodes.

As information from further application user sessions are obtained the PF tree can be modified to reflect the usage of the entity set by these applications. This example demonstrates how the PF operations can be used to maintain both the tree structure and the usage information.

[5] Note that for the purpose of this example the size of the fragment is measured in entity instances, assuming each entity instance is of the same size.
[6] Calculated as the number of primitive predicate accesses for the leaf node multiplied by the size of the fragment.

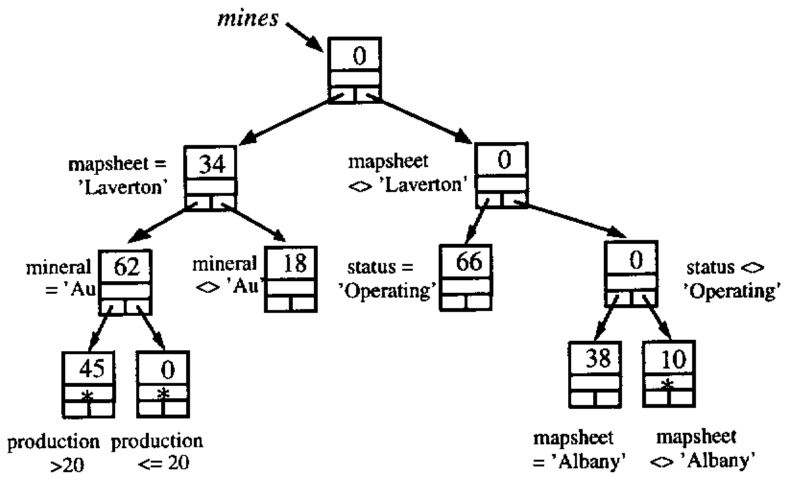


Figure 3.19 PF tree after pruning

3.5 Conclusions

The PF model provides an elegant and efficient means for capturing and maintaining application information. Such information is obtained from user sessions and used to construct a PF tree for each entity set in the database. The basis for the PF tree is a predicate. Predicates associated with usage information are identified from the applications and inserted into the PF tree. Primitive predicates can be identified from the tree and define the cells which are used in the fragmentation process.

A pruning process is used to identify and remove fragments that are not valid and provides a means of optimising the tree. To manipulate the PF tree, insertion and pruning operations are necessary. These operations - *reselect, aselect, nselect* and *delete* - are described and algorithms presented for their implementation. An example was provided which demonstrated how the PF operations are used to maintain both the tree structure and the usage information.

4 Geographic Data Distribution Methodology

4.1 Introduction

A methodology for geographic data distribution design is outlined in this chapter. The methodology follows a three-stage process to identify the distribution requirements, determine the fragmentation scheme and allocate the fragments to partitions at computer sites. The implementation of the methodology and the testing strategy is then outlined for the GEODDIS system.

Data distribution is not a static process. As application tasks and data access patterns change within a dynamic environment, data will have to be redistributed across sites. A strategy for dynamic data migration is detailed in the methodology.

4.2 Overview of the Distribution Methodology

The process of geographic data distribution takes a database as input and for output produces data partitions assigned to computer sites. It involves a sequence of decomposition, aggregation and placement as depicted in Figure 4.1. Entity sets are decomposed using fragmentation methods into fragments. The PF tree (see Chapter 3) is the basis for performing the decomposition. The fragments are then aggregated to form partitions and finally the partitions are placed at computer sites. The process requires database, application and site information (see Section 2.4.6).

The methodology used to perform the distribution of geographic data is based on a three-phase process that takes a global schema and produces a local schema for each computer site (Figure 4.2). The first phase is *distribution requirements analysis* and is concerned with obtaining and analysing the necessary database, application and site information. This information is used in the *fragmentation design* stage to identify valid fragments along with their associated usage information. In the *allocation design* phase the fragments are assigned to partitions and allocated to produce a local schema for each site.

The local schemas can change as the global schema and distribution requirements are altered. Therefore this methodology involves an iterative process to incorporate dynamic data distribution. As new information is obtained in the first phase a new fragmentation and allocation design are generated. The local schema is then modified at each site using a dynamic data migration strategy to reflect the new data distribution. The following sections expand on the details of the three phases of the methodology.

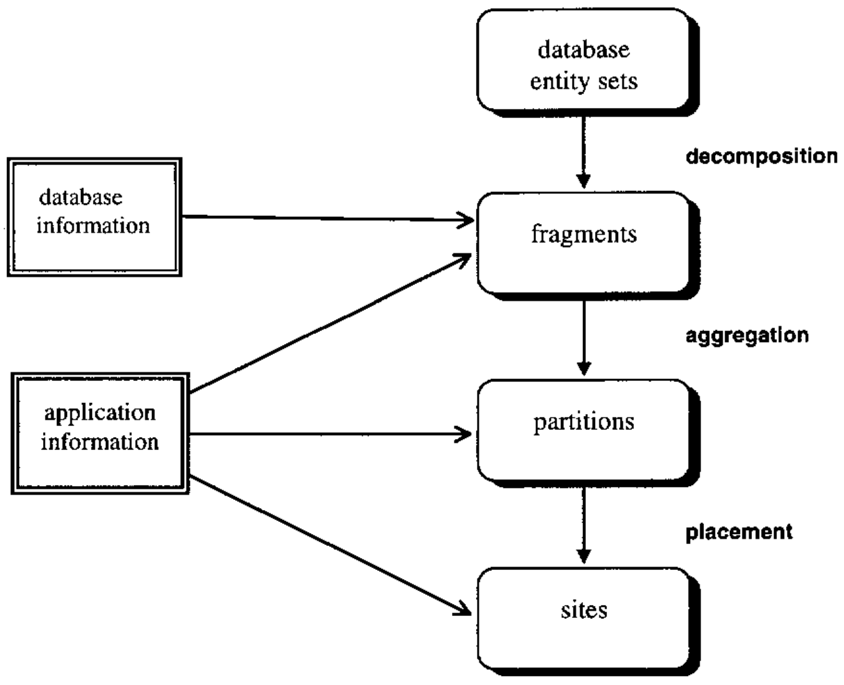


Figure 4.1 Process of geographic data distribution

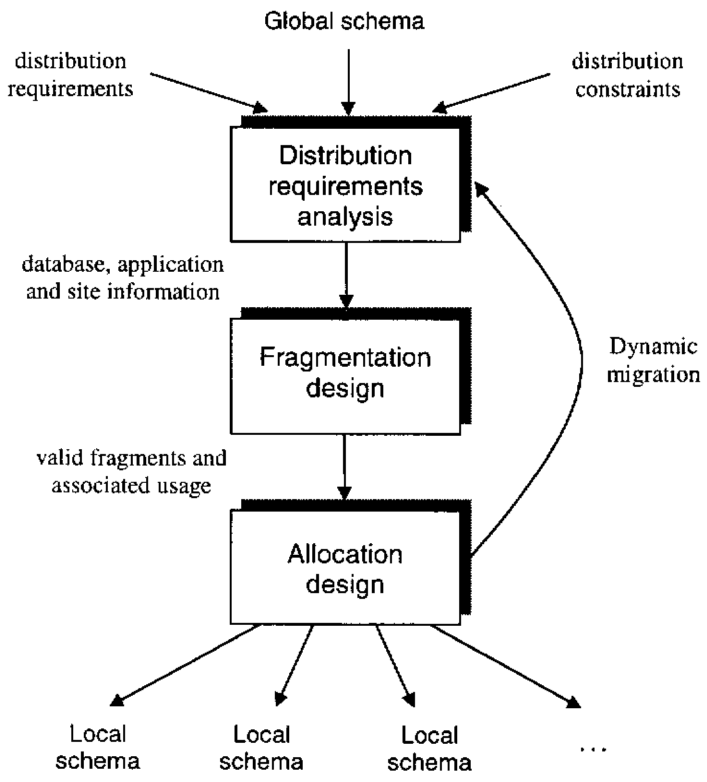

Global schema
distribution requirements
distribution constraints
Distribution requirements analysis
database, application and site information
Fragmentation design
Dynamic migration
valid fragments and associated usage
Allocation design
Local schema
Local schema
Local schema
...

Figure 4.2 Geographic data distribution design methodology

4.2.1 Distribution requirements analysis

The first phase involves the identification and collection of database, application and site information. The *database information* is obtained from the entity schema as well as the entity sets themselves (Section 2.4.6). It is obtained prior to the design, and must be updated when the entity schema or entity sets are modified. Information about entities that are created, deleted or modified can be gleaned from the application logs. This information is used to trigger the fact that the database information may need to be updated.

The database information required is:

- Number of entities,
- Relationships between entities, and
- Size of each point - in bytes.

For each entity the following information is required:

- Entity name - The name is used to identify the entity within application queries.
- Type of entity - The entity type is defined as either non-spatial, point, line or polygon.
- Cardinality - number of instances in the entity,
- Number of points in entity (for spatial entities) - This may be different from the cardinality if the spatial data type is not *points*. For example, the cardinality of a polygon entity would refer to the number of polygons.
- Number of attributes.

For each attribute, the information required is:

- Name of attribute - The name is used by application queries to specify the attribute.
- Size of attribute - in bytes.

In some cases, usage information is provided in terms of the number of entities accessed. For spatial data this may mean number of points, number of lines or number of polygons since the different spatial entity types occupy different amounts of storage. The number of *points* (of which each entity is comprised), which can then be easily translated into bytes, would be a better measure of the size of the spatial component of the entities (Sloan *et al.* 1992). However, if the number of points is unavailable then the number of entities multiplied by the average number of points per entity type, can be used as an estimate of the entity size.

Note that the size of an entity is determined by both the size of the spatial component and the size of the attribute component.

Application information (see Section 2.4.6) is extracted from session logs of users and applications. These logs keep track of the operations executed, entities affected, parameters used, type of entity access (*ie.* retrieval, update or both) time of operation and site at which operations occur. Parameters may involve a predicate expression to identify the entity instances required or it may involve some values obtained by user interaction. User interaction is often used especially with graphic and map data and involves mouse clicks and movements to identify such information as entity instances and geographic coordinates. For example, the user may click on two coordinate locations that identify a rectangular area defining the geographic data of interest. The coordinates identifying the geographic extent of the data must be recorded in the logs. This information can then be used in defining a selection of the entity set.

In particular, the following application information is required:

- Number of operations used, and
- Number of application sessions.

For each application session, the following information is necessary:

- Number of operations executed,
- Frequency of execution of each operation,
- Site at which session is run.

For each operation, the information required consists of:

- Name of operation,
- The number of entity sets accessed by the operation,
- The number of entity instances selected for each entity,
- The number and names of attributes accessed,
- Specification of which entity instances are accessed (*ie.* defined by a primary and/or derived predicate(s)),
- Site at which operation is executed (if different to the site at which the session is run),
- Type of access for each entity set (*ie.* retrieval, update), and
- Timestamp - records when operation is executed and data accessed.

Site information concerning data storage requirements and constraints must be identified by the Database Administrator before data distribution and updated as changes occur (Section 2.4.6). Site information is comprised of:

- Identification of sites that may contain data,
- Storage capacity at each site, and
- Any limitations or storage requirements for specific data at each site.

If specific data is limited to storage on one site only, then it is not included in the distribution design process. If it is limited to more than one site, then it is included in the process but the sites to which it (or its fragments) is to be allocated will be limited.

4.2.2 Fragmentation design

The second stage involves fragmentation design and encompasses the *decomposition* step shown in Figure 4.2. The database and application information obtained in the first stage is used to transform entity sets into cells. This is accomplished using the PF model as shown in Figure 4.3.

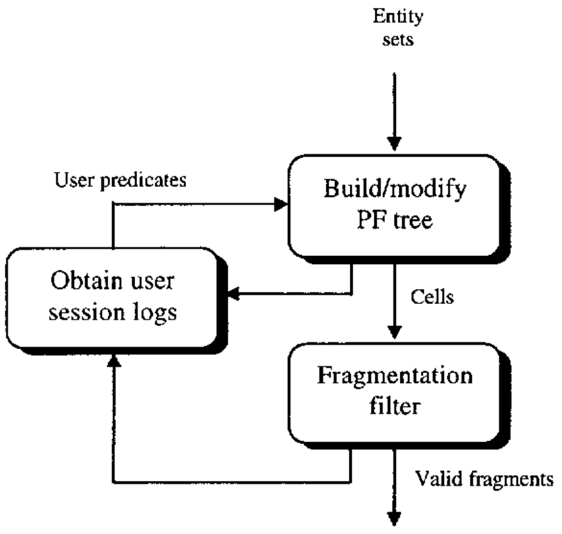


Figure 4.3 Using the PF model for *fragmentation design*

The primitive predicates and associated operations and usage information are extracted from user logs using simple string manipulation methods. Simple predicates are identified from the user predicates and inserted into the PF tree for the appropriate entity using the relevant tree insertion operations (see Section 3.3.1). The usage information is maintained in the PF tree for the predicates and primitive predicates (see Section 3.4).

The fragments identified by the primitive predicates may be empty, small, infrequently accessed or not recently accessed, and would therefore constitute *invalid* fragments as explained in Section 3.3.2. A fragmentation filter step (also explained in Section 3.3.2) is used to prune the tree so that only valid fragments are identified. These fragments can then be used in the *aggregation* step identified in Figure 4.1.

The fragmentation design is iterative as additional user session logs are identified, the PF trees are modified and the filtering process is executed. The user session logs should be processed in the order in which they actually were generated so as to reflect the usage by the applications. The fragmentation filter step need not be executed after every tree modification or even after every user session log is processed. It depends on how often the data usage patterns of the applications change which in turn influences tree growth.

Of course tree growth is dependent on the number of new predicates being inserted, as explained in Section 3.3.2. A tree grows particularly fast for selections made on a *current selection* containing the entire entity set. Much of the growth is often attributed to empty cells. This is because new cells are created with the assumption that a new predicate may involve portions of every existing fragment. However since user predicates are used to narrow down a selection of entity instances to only a subset of an entire entity set, only a subset of fragments will be affected. The cells associated with the remaining fragments are therefore empty and the corresponding nodes in the tree can be pruned.

After substantial tree growth, pruning is used to reduce the tree size and improve efficiency in maintaining the tree. Substantial tree growth refers to the addition of between 500 to 1000[7] or more primitive predicates after which the processing time for manipulating a PF tree increases dramatically, in fact, a potentially exponential growth if no pruning occurs (see Section 5.2 for detail on tree performance). If it is assumed that a user session results in a substantial tree growth, it is reasonable to prune the tree after every user session. It is also logical to prune between processing user sessions since a user session would generally refer to the same predicates and same fragments numerous times. However, because of variations in the number of different predicates used within user sessions, pruning may be done after a number of small sessions are processed, or even in the middle of processing a large user session. Empirical evidence used in this research suggests that pruning should occur after processing a typical user session of approximately 350 commands consisting of 30 user predicates in 60 selection operations.[8]

[7] After the addition of from 500 to 1000 nodes, the tree can potentially grow very quickly, running into the thousands and tens of thousands of nodes. Empirical evidence, based on the tests performed for this research, indicate a substantial decrease in performance in processing trees of this size.
[8] This estimate is obtained from a series of randomised user sessions that were used in the testing process. The actual user sessions contained an average of 336 operations consisting of 30 user predicates in 55 selection operations (see Section 5.2).

4.2.3 Allocation design

The valid fragments identified from the fragmentation design stage do not constitute the final fragmentation but must be aggregated into partitions to be placed at computer network sites. This is accomplished by the allocation design stage that identifies the partitions and the sites to which they are assigned (Figure 4.4). An *allocation strategy* determines an optimal site placement for each valid fragment. All the fragments assigned to a particular site are then aggregated to form a partition for that site.

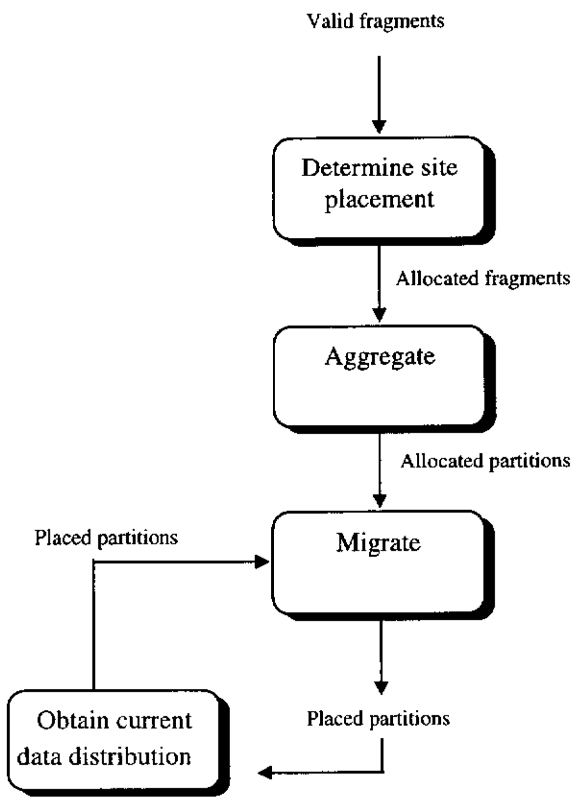


Figure 4.4 Aggregating and partitioning for *allocation design*

The final step is to migrate the data according to the partitioning strategy. This requires obtaining the current data placement and using it to obtain a data migration strategy. This process is detailed in Section 4.2.4.

In defining an allocation method, a known allocation strategy referred to as *locality allocation* was first considered in the context of geographic data. This strategy identified the data and method required for allocating fragments to sites based on *data locality* and using polarisation information for each operation (see following section). This is then compared to *site access allocation* where the PF model is used to provide the information for allocation based on frequencies of accesses at the various sites. Both methods are based on data locality and produce the same result but differ in the methods for arriving at an allocation. The PF model provides an eloquent and effective means of determining an allocation based directly on usage information obtained from user applications.

4.2.3.1 Locality allocation

The locality allocation strategy is based on how the data usage is polarised towards the network sites. This method has been defined by Ceri and Pernici (1985) for textual data and has been adapted for geographic data allocation in this research. Veenendaal 1994b also reviews this method. The information required as input to this allocation consists of:

- polarisation - the relative frequency of accesses to each fragment by each operation at each site,
- predicate selectivity of the fragment,
- number of records being accessed for the fragment, and
- frequency at which each operation accesses each entity at each of the network sites.

Polarisation refers to the relative frequencies of the data fragments of an entity being accessed by each individual operation at each network site (see Section 2.2.3). This information must be gathered for each fragment that has been identified from predicates defined on the entities of a database. The result is a 3-dimensional matrix that is aggregated by summing the values for each operation to produce a table showing the relative number of accesses of each predicate at each site.

Table 4.1 provides an example of the polarisation information of MAPSHEET by MAPSHEET_NAME for the *ARCS* operation[9]. The "*Laverton*" mapsheet is the predicate accessed primarily (80%) at Site 1 whereas Site 3 and Site 4 do not access this mapsheet at all. The "*Albany*" mapsheet is accessed mostly at Site 2. The third entry in the polarisation table indicates that all the mapsheets (represented as "*") are accessed 60% of the time at Site 4.

In addition to the polarisation data, information regarding the frequency of occurrence of each operation instance is required. Table 4.2 gives an example of frequencies for some GIS operations at four sites. Note that the operations are qualified by the entities that they deal with.

[9] The *ARCS* operation displays the lines of a given dataset, in this example the mapsheet boundaries.

Table 4.1: Polarisation of MAPSHEET by MAPSHEET_NAME

Predicate	Operation *ARCS*			
	Site 1 (%)	Site 2 (%)	Site 3 (%)	Site 4 (%)
mapsheet_name = 'Laverton'	80	20	0	0
mapsheet_name = 'Albany'	10	60	30	0
mapsheet_name = *	10	30	0	60

Table 4.2: Frequency of occurrence of some operations on four sites

Operation	Frequency of Occurrence			
	Site 1 (%)	Site 2 (%)	Site 3 (%)	Site 4 (%)
points *rivers*	25	25	30	20
polygonshades *mapsheets*	30	30	20	20
intersection *mines mapsheets new*	50	20	10	20
build *new*	25	30	20	25

Further information required for the locality allocation is the predicate selectivity which indicates the relative size of the predicate with respect to the entity, and the number of records accessed by each operation on the entity. This information is obtained in the distribution requirements analysis stage.

The allocation is based on the number of relative accesses for each operation on each fragment at each site. The relative access values are calculated as follows (Ceri and Pernici 1985):

$$R_{ijk} = PS_i * OF_{ijk} * PA_{ij} * Po_{ijk},$$

where R_{ijk} = relative access value for operation *j* on predicate *i* at site *k*,

PS_i = predicate selectivity for predicate *i*,

OF_{ijk} = frequency of operation *j* on predicate *i* at site *k*,

PA_{ij} = number of record accesses of operation j on predicate i, and

PO_{ijk} = polarisation of predicate i for operation j at site k.

The relative access value R_{ijk} is calculated for each predicate (which forms a cell) with each operation on each site resulting in a 3-dimensional matrix. The effective locality access values for each cell on each site can then be calculated:

$$L_{ik} = \sum_{j=1}^{n} R_{ijk} ,$$

where L_{ik} = locality access value for predicate i on site k.

Each cell can then be assigned to the site with the highest locality access value.

$A_i = \max_k \{ L_{ik} \}$ (see footnote [10]) ,

where A_i = site allocation for predicate (cell) i .

Given the site allocations for each cell, the partitions can then be determined for each site.

The locality allocation method was implemented and an important drawback was identified. The problem is the difficulty in acquiring the large amount of usage data that must be collected. For every combination of operator, predicate, and site, the information to be collected and stored includes polarisation and frequency of occurrence, the predicate selectivity for each predicate for each entity, and the number of record accesses for each combination of operation and predicate. For n operations, m sites, and p predicates, the amount of information required to be stored as observed as:

$I = 2nmp + p + np$,

where I refers to the number of records of usage information.

Often many operations access the same data or data fragment. Collecting information for each operation means that for a particular data fragment, usage data is dispersed across the information for each operation that executes on that fragment. Only when a data partitioning occurs is that data aggregated to determine the usage of that fragment. The amount of usage information maintained overall is therefore quite high and therefore also difficult to manage.

To overcome these difficulties, the distribution methodology uses the *site access allocation* method to determine an allocation based on the usage properties of predicates in the PF tree.

[10] We use $\max_k \{ N_k \}$ to refer to the maximum value of the k values in the set.

Only the predicates that define valid fragments are maintained and only usage information pertaining to these fragments at each site are accumulated. Information about individual operations need not be maintained; instead, the resulting usage properties are accumulated in the PF tree structures. Therefore the amount of information required is substantially reduced and for *m* sites and *p* predicates is estimated as:

$$I = mp$$

The *site access allocation* method is further discussed in the following section.

4.2.3.2 Site access allocation

Using the site access allocation strategy, fragments are assigned to the site at which they are most frequently accessed. The PF tree is used to identify the cells and obtain site access frequency information for each. The allocation is determined by the highest frequency of access as follows.

$$A_i = j,$$

where $\max_j \{ SF_{ij} \}$,

A_i = site allocation for fragment i, and

SF_{ij} = site access frequency for fragment i at site j.

To aggregate the fragments, the associated primitive predicates must be combined into one predicate expression to identify the resulting partition. This is accomplished by taking the disjunction[11] of the primitive predicates.

$$PC_j = \bigvee_{i=1}^{n} PP_i \text{ for } A_i = j,$$

where PC_j = predicate clause defining the partition at site j, and

PP_i = primitive predicate describing cell i.

A situation may arise where two or more fragments are allocated to the same site because of similar usage properties. For such a case the aggregation process would effectively result in undoing the fragmentation. This may mean that some predicates defining these fragments can be discarded and some primitive predicates may be merged to form a new one. This is

[11] The notation used to represent the *disjunction* is the $\vee$ operator used as follows:

$$\bigvee_{i=1}^{n} PP_i = PP_1 \vee PP_2 \vee, \ldots, \vee PP_n$$

essentially one of the pruning methods, namely, the pruning of fragments allocated to the same site (Section 3.3.2).

As an example, consider the PF tree of Figure 4.5 (a) obtained from user information for entity *mapsheets*. Three fragments are defined by the primitive predicates and have been allocated to sites 2 and 5. However, two of the cells defined by the primitive predicates "*mapsheet = 'Laverton' AND mineral = 'Au'*" and "*mapsheet = 'Laverton' AND mineral = 'Au'*" have been allocated to site 2. These fragments will therefore be merged. Further, the predicates defined in the leaf nodes share the same parent. By removing these predicates, the two primitive predicates can be merged to form one, namely, "*mapsheet = 'Laverton'*" as shown in Figure 4.5 (b).

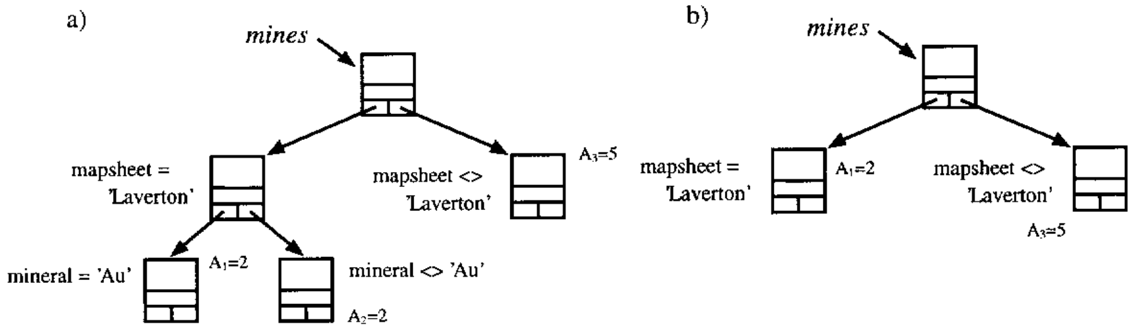


Figure 4.5 Using the PF tree for aggregating fragments: a) before aggregation b) after aggregation

In general, any nodes whose left and right subtrees are allocated to the same partition can be pruned using the PF *delete* operation. The remaining parent receives the site allocation instead. For each site, the disjunction of the resultant primitive predicates allocated to the site define the partition for that site.

4.2.4 Dynamic data migration

Data placement is certainly not a static process but changes over time as data usage and locality changes. Data migration essentially involves a reorganisation of fragments among partitions. However the fragments themselves may be altered, old fragments may be deleted and new ones may be defined. In addition, some partitions may be removed and additional ones created.

Recall that a partition P_i is a set of fragments:

$$P_i = F_{i1} \cup F_{i2} \cup, \ldots, F_{i2}, \text{ where } F_{ij} \text{ is fragment } j \text{ found in partition } i .$$

The result of the migration process is a new set of partitions P^* defined from the old set P where the number of partitions in each set may be different. This difference is due to changing circumstances such as adding new computer sites that provide storage space for more partitions. The migration strategy used is outlined with the following pseudo code.

```
for each $P^*_j$
    for each $P_i$
        if $(j \neq i)$
            for each fragment $F^*_{ip}$ in partition $P^*_i$
                for each fragment $F_{jq}$ in partition $P_j$
                    $I = F^*_{ip} \cap F_{jq}$
                    if $I \neq$ NULL
                        move I to partition $P^*_j$
                    endif
                endfor
            endfor
        endif
    endfor
endfor
```

The algorithm essentially involves comparing all N old fragments with all M new fragments and is therefore of order M*N in complexity.

What makes the migration process more complex is the fact that the fragments themselves may be re-fragmented by splitting and/or merging. The intersection (I) between the old and new fragment is used to identify the portion of the old fragment that must be migrated to the new partition as part of a new fragment. There may be a number of such portions of fragments (*ie.* fragment of a fragment) which make up the new fragment F_{ip}^*. Note that these portions may come from different fragments in the old partition.

For example, in Figure 4.6 a new fragment F^*_{j2} in the new partition is constructed from all of fragments F_{i3} and F_{i4} and from portions of fragments F_{i1} and F_{i2} from the old partition. Of course (portions of) fragments from different partitions may be combined into one fragment in the new partition.

The PF model is used to identify the new fragments F^*_{ip} and their corresponding partitions P^*_i. As usage and data information are accumulated by the applications, the PF tree is altered to reflect the changes. Not only are the partitions altered by an aggregation of fragments, caused by pruning, but fragments themselves may be altered due to additional predicates being used within subsequent applications. The PF tree that contains this information is therefore used to identify the new partitions.

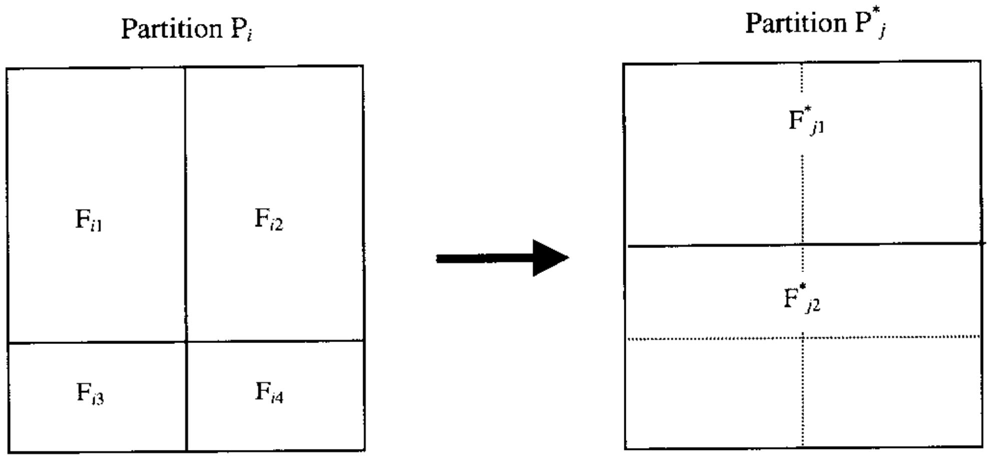
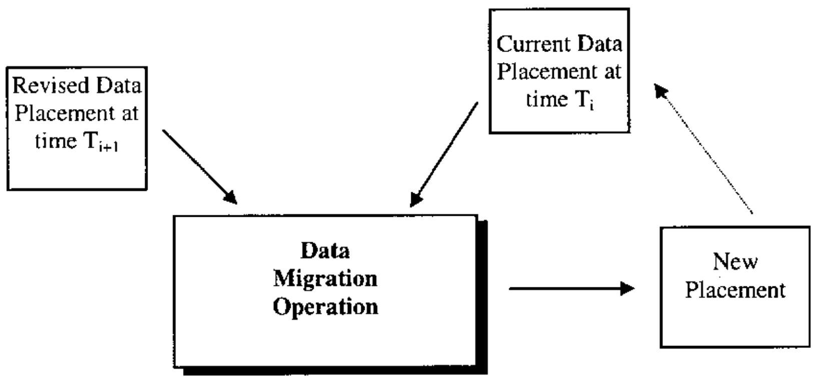


Figure 4.6 Constructing a new partition by re-fragmenting an old partition

When a data placement is performed a snapshot of the current PF tree configuration indicates the current placement for an entity set. This information, along with a snapshot of a revised PF model reflecting subsequent usage information, is used to determine a data migration strategy based on the algorithm shown above. This new placement then becomes the current placement in a subsequent migration operation (Figure 4.7).



Revised Data Placement at time T_{i+1}
Current Data Placement at time T_i
Data Migration Operation
New Placement

Figure 4.7 Data migration process using current and revised placements

The current placement of an entity is maintained as a PF tree. Thus two trees are maintained for each entity, one which identifies how the entities are currently placed and another which is accumulating more recent application information. When a data migration occurs the current data placement subsequently becomes the new data placement which now reflects the current PF tree structure.

A further issue involves the frequency at which data migration should occur. Because it is an expensive operation, it should not occur too frequently (see Section 2.4.5). However it must occur often enough to ensure that the data distribution does not become out of touch with the current applications and their usage. The migration frequency depends on the amount of change in data access patterns.

Data migration should be undertaken when the benefits of the new data placement exceed the costs of performing the data migration. The benefits of the *new* data placement are determined by the redistribution cost plus the difference in cost of applications accessing data using the *old* placement (which is the current placement) versus the cost of applications accessing the data using the new placement. This difference can only be quantified by querying the database to obtain the cost of data access. The applications used must be representative of current applications and may involve different entities and predicates than were accessed from a previous placement. The data required by the application queries must be identified from both the old and the new placements by executing the queries on the fragments of these placements. The cost of access for each of these placements can then be determined and the reduction in cost (if any) identified. The manner in which data access costs are determined is explained in Section 4.3.

This reduction in cost, which identifies the benefit of the new data placement, can then be combined with the cost of redistributing the data. The entities and predicates for the new placement must be compared to the entities and predicates for the old placement. Once again this is accomplished by querying the database. The queries involving the predicates must be executed on the database and the entity instances contained in their intersection identified. The data migration cost is determined from those entity instances that will reside on a new site according to the new placement. If the data migration cost is less than the reduction in cost based on the new placement then the data migration process can be executed. Note that there are substantial overhead costs involved in querying the database for determining both the data access and migration costs.

Quantifying the costs involved in data access and migration assists in determining whether or not migration should take place, but it does not determine the frequency of migration. Even worse, it does not consider the frequency at which a potential data migration should occur and hence it is difficult to determine how often to obtain the costs for evaluating a migration. Of course the costs could simply be obtained at regular intervals, but the high overheads in obtaining these costs make that prohibitive. An alternative would be to monitor the performance of actual queries, but this would mean that additional log information regarding processing and communication costs would need to be captured. Determining the frequency of data migration must therefore involve a Database Administrator who will have to use his/her experience and knowledge of data, usage properties of data, and hardware/software (*eg.* regarding costs of access). Note that the latter may change with advances in hardware and software technology (Strand 1999). The Database Administrator may effect a data migration in the following situations:

- There is a marked or known change in data usage and access characteristics,
- After a series of application user sessions which have similar data usage characteristics, or
- When the cost of improvement in data access exceeds the cost of data migration. Such costs can be obtained infrequently at regular or random intervals.

4.3 Measures of distribution

In order to compare distribution strategies, metrics must be developed to account for the costs and benefits of each strategy.

To understand the costs associated with data distribution, consider the example in Figure 4.8. Entity *mines* is distributed across three partitions at three sites: A, B and C. Suppose an operation called *build_topology* is executed by an application at site C and performs an update operation on the entire entity.

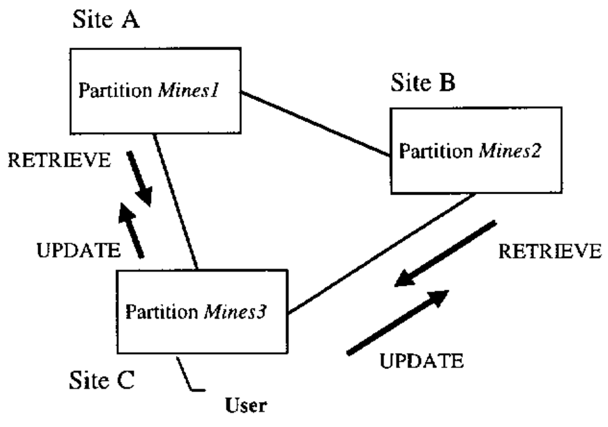


Figure 4.8 Retrieval and update costs associated with data distribution

Since not all the data required by the operation is found at the local site (site C) where the operation is being executed, some data transfer operations will first have to be initiated to retrieve the required data from remote sites. In this example, the fragments for entity *mines* at sites A and B must be transferred to site C where the operation is being executed. This retrieval involves a transfer cost based on the amount of data to be transferred. Similarly, if the data (and hence the partitions) require updating, then an update cost is incurred. The data

$$DC_p = \sum_i \sum_j \left(RC_{ij} + UC_{ij}\right),\ A_{ij} \neq B_p\ ,$$

distribution cost for executing an operation can be formulated as follows:

where DC_p is the data distribution cost for operation p,
RC_{ij} is the cost of retrieving fragment j of entity set i,
UC_{ij} is the cost of updating fragment j of entity set i,
A_{ij} is the site allocation of fragment j of entity set i, and
B_p is the site at which operation p is executed.

Further costs are incurred by the data distribution. Before an operation can execute on an entity set, the individual fragments that have been retrieved may first have to be merged (Figure 4.9) involving processing costs. Similarly, when the entity set is updated the partitions will have to be re-created by splitting the entity set. The total cost associated with an application operation therefore depends on whether the operations involve retrievals and merges, updates and splits, or both.

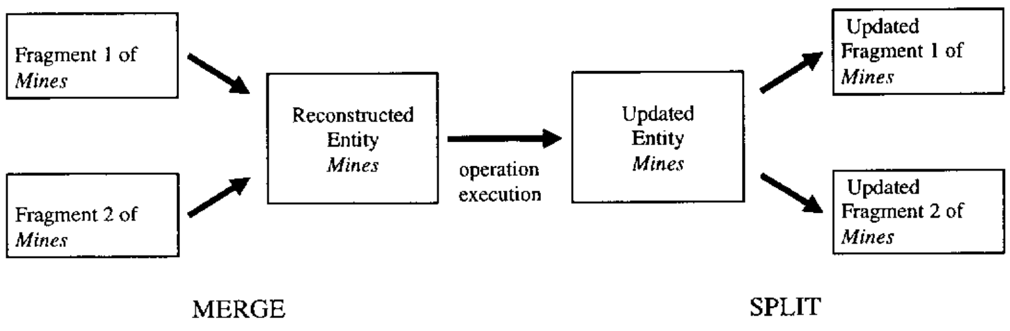


Figure 4.9 Merging and splitting costs associated with data distribution

These retrieval/update and merge/split costs are related to the number of data instances of an entity. For spatial point data these costs are correlated directly with the number of entity instances (*ie*. points) for that entity. However, line and polygon entity instances vary in complexity with polygons containing a variable number of lines and lines constructed from a variable number of points. Rather than being related to the number of entity instances, a better measure of cost would be the number of points that make up the line or polygon entity (Sloan *et al.* 1992).

Both the spatial data and their attributes must be considered in determining the size of a fragment. The attributes are associated with the entity instances which means that the number of attributes and number of entity instances must be known in addition to the number of points. The size of a fragment can therefore be determined in bytes as follows:

$$\text{Size}(F_{ij}) = \text{Numpts}(F_{ij}) * \text{Point\_size} + \text{Numents}(F_{ij}) * \text{Att\_size} \ ,$$

where Numpts(F_{ij}) is the number of points in fragment i of entity j,
Point_size is the number of bytes used to store each point,
Numents(F_{ij}) is the number of entity instances in fragment i of entity j, and
Att_size is the number of bytes required to store the attributes for one entity instance.

Using the number of points as the basic measure of size for spatial data fragments, the retrieval/merge and update/split operations involving data transfer and processing costs can be estimated as follows[12]:

$$RC_{ij} = (1 + c_1)\text{Size}(F_{ij}) \ ,$$
$$UC_{ij} = (1 + c_1)\text{Size}(F_{ij}) \ ,$$

where c_1 is a constant relating the processing cost to the data transfer cost. This constant value is a means of normalising the costs so that they both can be estimated from the fragment size. The value is, of course, dependent on the data transfer speed over the computer network relative to the processing speeds at the site where the data is processed. It is estimated by a Database Administrator using knowledge about the given hardware specifications and from empirical evidence. It is assumed that this value can be estimated and the estimate is known. Note that a fragment that is accessed and subsequently updated incurs both a retrieval cost and an update cost.

Application operations may, of course, involve more than one fragment. The costs associated with retrieving and/or updating are incurred for each fragment accessed by the operation. These costs, however, are only an estimate. For small fragments the costs would likely be higher relative to the fragment size and due to the overheads of data distribution. For multiple and consecutive operations accessing the same fragment, these costs may be lower due to buffering and caching which may reduce the amount of data being retrieved.

For application operations that require access to only one fragment, it is obvious that merging and splitting costs are not incurred. This means that the processing costs do not

[12] Note that because the number of data points is used as the basis, the retrieve and update costs are estimated to be equivalent. However, rather than using one variable to refer to both costs, we will continue to identify them separately so as to make clear the nature of the costs involved.

need to be factored into the retrieve/update costs. This is achieved by setting the constant value c_1 to be zero (0).

As already discussed in Section 3.3.2 a threshold value needs to be set to determine when pruning occurs for infrequent access and small fragments. A fragmentation is beneficial only if the cost of fragmentation is less than a threshold value which is equivalent to the cost of leaving the data unfragmented. The threshold value τ is estimated as follows:

$$\tau = RC_{ij} * (RF_{ij} + RF_{ij+1} + RF_{ij+2}) + UC_{ij} * (UF_{ij} + UF_{ij+1} + UF_{ij+2}) \ ,$$

where RC and UC are the retrieval and update costs,
F_{ij} is the fragment being considered for further fragmentation,
F_{ij+1} and F_{ij+2} are the resulting fragments of the potential fragmentation of F_{ij}, and
TF_{ij} is the total frequency of accesses for fragment i of entity j.

This threshold value is obtained from the PF tree by examining the access information of a node (*ie.* the original fragment) and its two children (*ie.* the two fragments of the parent). The cost of the fragmentation υ can then be calculated as follows:

$$\upsilon = (\sum_{k=0}^{2} (RC_{ij+k} * RF_{ij+k}) + (UC_{ij+k} * UF_{ij+k})) + UC_{ij+1} + UC_{ij+2}$$

In this calculation both the retrieval and update access frequencies and fragment size information are examined for the parent and two children nodes in a PF tree. The last two terms in the expression take into account the cost of setting up the fragmentation in the first place. This cost involves an "update" cost as the two new resulting fragments are placed at their respective site locations. The fragmentation is beneficial if the cost of fragmentation is less than the threshold value.

Consider the two examples of Figure 4.10 where the fragmentation of "*mapsheet = 'Laverton*" is being considered. The PF tree is constructed, assuming that the fragmentation does take place, and the number of retrieval and update accesses (shown in the nodes in the figures) are then accumulated for each fragment. Assume that the total retrieval and update costs accumulated thus far are those as shown in the figure. Assume also that the processing and transfer speeds used in obtaining the retrieval and update costs are equivalent (*ie.* $c_1 = 1$). This means that when considering the case where *mapsheet = 'Laverton'* is not fragmented, we can assume that the update and retrieval costs are halved (*ie.* equal to 140) since there are no merge or split costs involved in accessing the data.

The threshold value and cost of fragmentation are calculated in example a) as:

τ=140*(18+93+35)+140*(5+32+8)=26740

υ=280*18+280*5+90*93+90*32+190*35+190*8+90+190=26140

Because the cost of fragmentation is less than the threshold value, the fragmentation is considered beneficial and therefore is maintained. For part b) the threshold value is calculated as τ=23800 and the cost of fragmentation as υ=33220. As a result the fragmentation is not beneficial and pruning must be carried out to remove the relevant nodes from the PF tree.

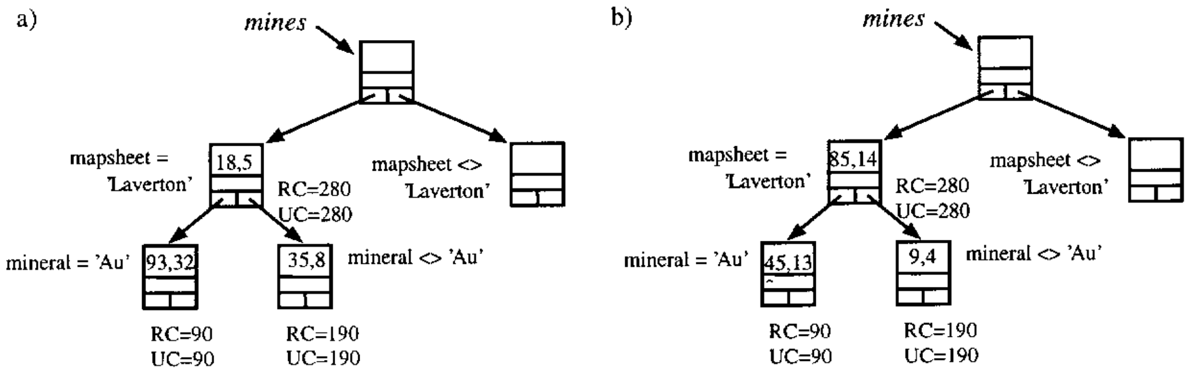


Figure 4.10 Examples for determining the cost of fragmentation

The amount of fragmentation can be measured as the average number of fragments per entity and is calculated as:

$$FI = \left(\sum_i f_i\right)\Big/ n \ ,$$

where FI is the fragmentation index,

f_i is the number of fragments for entity i ,and

n is the number of entities in the database.

An index value of one (1) indicates that no fragmentation exists for any entity. The higher the index the greater the average fragmentation among the entities in the database. The fragmentation index, while providing a general measure of how much a database is fragmented, does not indicate if the fragmentation is evenly distributed across all entities or is concentrated on a small proportion of the entities.

The amount of fragmentation could also be due to a large number of cells identified from overlapping application queries. This also means that many fragments may have to be accessed to satisfy one application query. The overhead incurred in accessing and merging many fragments may dictate that the cost of fragmentation is too high and should be reduced. The amount of overlap among application queries can be measured using the following calculation:

$$OI = \left(\frac{\sum f_i}{\sum u_i} \right) \Big/ n \ ,$$

where OI is the overlap index, and

u_i is the number of user fragments relating to entity i.

An overlap index value of one (1) indicates that the number of fragments for distribution is equal to the number of user fragments. An increase in the index value means the user fragments are further fragmented because of overlap and hence there will be an increase in the cost of fragmentation.

These measures for data distribution provide an estimate of the costs involved in fragmentation and allocation. They are used in Chapter 5 to evaluate the methodology being proposed.

4.4 Implementation of Geographic Data Distribution

The implementation in the geographic data distribution system referred to as GEODDIS (GEOgraphic Data DIStribution) involves building a data model and generating algorithms for the steps of the distribution methodology. This is detailed in the following sections.

4.4.1 GEODDIS data model

The GEODDIS data model is constructed around the PF tree structure. Each node in a tree refers to a user predicate. Each user predicate is defined on one or more entity sets (*ie.* derived predicate - see Section 2.5.1) and is referred to by one or more application operations that can execute on one or more sites. All this information is maintained in the GEODDIS data model as illustrated in Figure 4.11. The arrows indicate the direction of the relationships between the components of the data model.

Each entity set in a geographic database has an associated PF tree. The predicate expressions that are associated with the tree nodes are contained in the *predicate* table. For each predicate expression, being a simple expression (see Section 3.3), the information maintained is two operands and a relational operator. An operand may refer to an entity attribute.

Information about the entity sets is provided in the *entity* table. The database information identified in Section 2.2.6 is stored in this table.

For each unique occurrence of a predicate expression and an application operation that refers to it, an entry is maintained in the *operation-entity* table. Because an operation may refer to more than one entity, information about which entities are accessed and any associated predicates is provided. Hence there are links from the operation-entity table to the entity and predicate tables. The name, number of entities it refers to and the type of access for each entity is maintained for each unique operation in the *operation* table. The operation-entity table also relates to the information contained in the *site* table by specifying the site(s) at which the operation is executed.

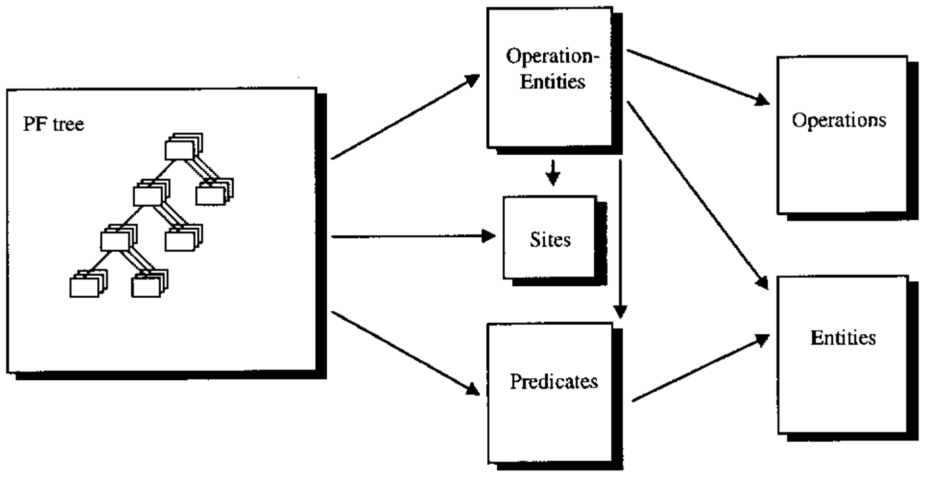


Figure 4.11 GEODDIS data model

Note that the operation and the operation-entity tables were maintained only to implement the locality allocation method (Section 4.2.3.1). They are not necessary for the site allocation method which is the preferred method for reasons explained in Section 4.2.3. The data model is more fully detailed in Appendix A.

As the application information is obtained, usage information for the predicates and entities is accumulated and maintained in the PF trees (see Sections 3.4 and 4.2.1).

4.4.2 Capturing application information

During the initial database design stage of a GIS application the entities, operations, and their schemas are identified and represented in GEODDIS. If the database is already in existence then the existing information is entered into GEODDIS. This constitutes the data information (see Section 4.2.1) required for data distribution.

The application information is obtained from *log* files of user application sessions that have been executed on the network sites. From this information GEODDIS produces a data distribution scheme that is then applied to the geographic database (Figure 4.12). Subsequent application sessions access the newly migrated data. As additional data and application information become available, a new data distribution scheme can be obtained and applied once again to the database. The process is iterative as would be expected in a dynamic environment.

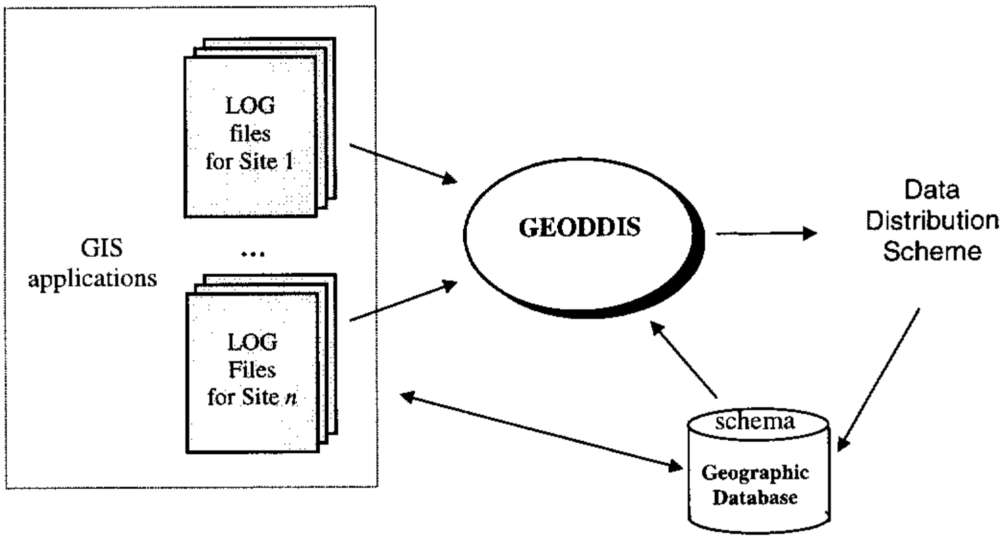


Figure 4.12 GEODDIS is linked to a GIS application

The GEOMINE application used to test the geographic data distribution methodologies of this research was implemented using the ArcInfo GIS (see Section 4.5 which explains the testing strategy). The entity schema information was obtained by executing ArcInfo database commands to list the details of the entity sets and store them in text files. Simple string manipulation techniques were then used to identify the entity names, geographic data types, attribute names, field sizes and the number of point features and entity instances for each entity set. The operation schema information could not be readily extracted from the ArcInfo GIS and had to be manually entered into GEODDIS.

ArcInfo provides a logging facility which logs all commands and operands used by application sessions. It also provides a *watch* facility which, when initiated, logs additional detailed information involving individual queries, predicate expressions, coordinate information entered either explicitly or through mouse or cursor operations and the number of entity instances selected by the various data selection operations. Simple string manipulation techniques are used to extract the operation, predicate and associated usage information necessary for data distribution.

The *watch* files in particular were very useful since they identified specific coordinate information for spatial selection and query operations involving on-screen mouse clicks or digitiser cursor input. For example, suppose a user wants to identify a rectangular geographic area from some data being displayed on the screen. The geographic region can be specified by placing and clicking the mouse on two opposite corners of the rectangle. The actual X and Y coordinates identified by the mouse clicks are captured by the *watch* file. GEODDIS can extract the coordinate information from the *watch* file and appropriately insert it into the associated predicate expressions involving the spatial (*ie.* X and Y) coordinates. In addition to identifying coordinate values the *watch* files also indicate the number of entity instances selected by a selection operation. This has proven to be very useful in determining the size of fragments identified by predicate expressions.

Information about entity sets and their schemas are also obtained from the log files. Information is obtained from operations involving entity set additions and deletions and schema modifications. The database information can therefore be dynamically updated as the application information is obtained.

4.4.3 Using GEODDIS in partitioning and allocation

Once the database and site information is obtained the application information can be extracted from the log files and inserted into the PF tree. This involves the tree building and fragmentation filtering steps of fragmentation design (Section 4.2.2). At this point the PF trees contain primitive predicates referring to potentially both valid and invalid fragments. For reasons explained in Section 5.2.2 pruning can take place immediately after a user session is processed. The primitive predicates defining empty fragments are therefore pruned.

The process of pruning empty primitive predicates requires that fragment sizes are known. They can only be obtained by using the primitive predicate expressions to query the database. This is achieved by building a query file containing the primitive predicates and executing the queries within the GIS database. The result is a file containing the number of entity instances selected for each query. The corresponding primitive predicates in the PF trees can then be updated with this fragment size before the pruning takes place. Note that in the pruning step the fragment size of the (new) primitive predicates is maintained. However in the case of insertions of new predicates this information is lost and therefore the database must be queried again, this time with the new primitive predicates, to obtain the sizes of the new fragments.

When an opportunity for data migration occurs (Section 4.2.4), after some user sessions have been processed, pruning using additional methods (Section 3.3.2) can occur. Predicates with a *last accessed* time prior to the previous data migration are pruned. Note that the Database Administrator can determine whether a time corresponding to a different time interval will be used. For example, for data that is periodically accessed, the time interval may be set to the second last data migration or an interval independent of previous data migration times. Further pruning takes place for small and/or infrequently accessed fragments using the threshold value as described in Section 4.3. The result is that only valid fragments remain as identified by the remaining primitive predicates in the PF model.

The next step in implementing the data distribution methodology is to allocate the fragments to sites based on their usage frequencies at each site. The result is that each primitive predicate in the PF model has a site allocation. If there is a requirement for the fragment to be stored on a limited number of sites, then that site with the highest usage frequency for that fragment is chosen.

At this point in the distribution design a data migration can potentially occur. As explained in Section 4.2.4, this process requires identifying the data based on the previous placement that must be migrated to a new site. The data is then migrated to the new site and the new placement details are stored by saving the primitive predicates of the PF trees along with their site allocations. This placement becomes the "old" placement for a subsequent migration operation.

The whole process can now iterate beginning with the processing of application logs. The implementation of this entire process in GEODDIS is summarised in the following algorithm using pseudo code.

```
loop for application
        for each data migration interval
                for each user session
                        read log file
                        update PF tree for each entity accessed
                        build list of queries for each primitive predicate
                        run list of queries in GIS to determine fragment sizes
                        update PF tree primitive predicates with fragment sizes
                        prune trees based on fragment size
                endfor
```

```
            prune fragments from PF trees based on infrequent access, time last used
                and fragment size
            allocate fragments to sites
            prune trees based on fragments allocated to the same site
            generate list of data migrations based on allocation and previous placement
            run list of data migrations for GIS
            save data placement
        endfor
    endloop
```

GEODDIS consists of operations to manipulate the PF tree and interact with a GIS database and applications. A detailed description and example of the syntax and command structure for GEODDIS is provided in Appendix C. GEODDIS is implemented on a SUN UNIX operating system platform using the C programming language and UNIX shell scripts. Appendix D provides an example of a shell script from which the C-language programs and the ArcInfo GIS are executed. The shell script uses the above algorithm to read in all currently existing application logs and perform a site allocation and data migration. Appendix E provides a structure chart of all the C-language program modules and their relationships to each other.

4.5 Testing strategy

The methodology explained in this chapter and implemented in GEODDIS was tested using the GEOMINE application developed in ArcInfo and introduced in Section 1.5.

User session information was obtained for the GEOMINE application. A series of nine individual user sessions, representative of a range of typical sessions, were logged in ArcInfo and used as the basis for evaluating the methodology. These user sessions were chosen since they vary in terms of the number and range of operations executed and available from GEOMINE. These operations access a range of spatial and attribute data available within the GEOMINE database. Each individual user session is described in Appendix B.

The user sessions are used individually to test and evaluate the PF model then collectively to test and evaluate the geographic data distribution methodology using the PF model. In order to provide a comparison the operations in all nine user sessions were randomised across a new set of nine user sessions. The same number of user sessions was chosen to maintain the average size of each user session. It must be noted that the order of the selection operations also changes in the randomised user sessions. This has the potential of generating very different results as explained in Section 3.3. The effect of ordering will therefore be tested by observing the randomised user sessions.

A cross-validation testing strategy was used to test and evaluate the methodology. Cross-validation involves using some of the data to *train* the system and using the remainder of the data to *validate* its performance (Mitchell 1997). For *k*-fold cross-validation this occurs *k* times using *k* disjoint subsets of the data. In this case some of the user sessions were used to generate a data distribution strategy and the remainder of the user sessions were used to evaluate this strategy. Rather than test with individual user sessions, it was decided to use a group of user sessions as is done in the methodology[13]. A 3-fold cross-validation approach was chosen involving three groups of three user sessions each. A data placement was determined for 2 groups of sessions that constitute the training data. The remaining group, the validation data, was then used to test the effectiveness of this placement (Figure 4.13). This was repeated for all combinations of the three groups. This testing strategy was used for both the original user sessions as well as the randomised user sessions.

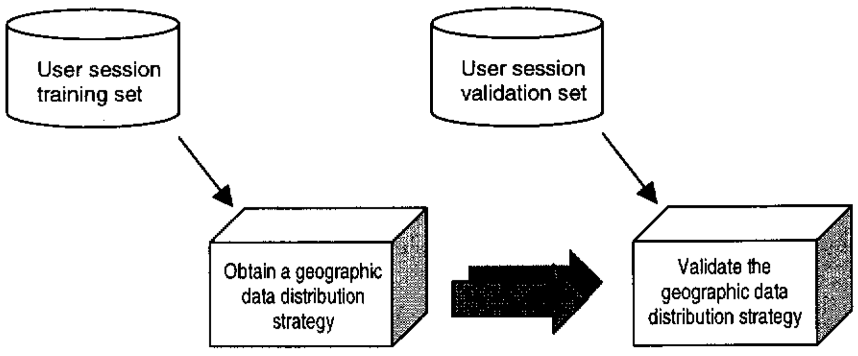

User session training set
User session validation set
Obtain a geographic data distribution strategy
Validate the geographic data distribution strategy

Figure 4.13 Testing strategy for data distribution methodology

These testing strategies are further described in Chapter 5 along with the results and discussions.

[13] Within the methodology a data migration occurs after processing multiple user sessions although it is possible to use a single user session before doing a data migration. As explained in Section 4.2.4 a data migration operation should not be executed too frequently and would not normally occur after a single user session was processed, but rather only when sufficient change in the data usage patterns warrant it.

4.6 Summary and Conclusions

The geographic data distribution methodology outlined in this chapter builds on previous work concerning data distribution. The query-adaptive approach, using information on application queries and associated data usage properties, is incorporated into the methodology in an effective manner using the elegant tree-based PF model designed for this purpose.

The methodology follows a three-phase process of distribution requirements analysis, fragmentation design and allocation design. The methodology handles geographic data and produces a distribution strategy involving procedures and cost metrics that cater for the spatial components of vector geographic data. The implementation in GEODDIS interacts with a GIS and uses the PF model to provide an effective and elegant means of collecting and maintaining information in supporting a geographic data distribution design strategy.

A number of cost measures have been developed so that the effectiveness of the fragmentation, allocation and data migration strategies can be determined. The measures developed are used to provide an estimate of costs which assist in determining whether or not a fragmentation is beneficial or a migration operation should occur. Database Administrator support is necessary to oversee these decisions and override them when additional and often non-quantifiable factors must be taken into consideration.

The methodology is dynamic and incorporates changes in the database and application environments. As application information becomes available and as the database changes, the fragmentation and allocation schemes can be altered to reflect the changes. This is important since GIS environments and data usage are not static.

5 Results and Discussion

5.1 Introduction

In evaluating the methodology for geographic data distribution, a number of aspects must be addressed. Firstly, the PF tree is the basis for the methodology and its performance and effectiveness in maintaining data usage information must be evaluated. As detailed in Section 5.2, pruning techniques are necessary for maintaining the efficiency of tree implementation. Further, the implementation is invariant to the effects of the ordering of user queries.

The second aspect involves the evaluation of the methodology itself and the use of the PF model in implementing a geographic data distribution. This evaluation is described in Section 5.2 and involves test results and conclusions for the various stages of the methodology involving fragmentation, partitioning and allocation, and data migration. Cross-validation testing is used to demonstrate that the distribution of data is improved over single site and random allocation strategies.

5.2 PF Tree Performance and Evaluation

The performance of the PF tree is a factor of its size and shape. The types of selection operations and their order determine the tree shape and size (Section 3.3). While tree growth is, of course, unavoidable it can be and is constrained by the actual number of fragments that are produced. Any predicates and primitive predicates in the tree that result in unnecessary fragments require pruning (Section 3.3.2). The use of pruning techniques for the PF trees ensures that they perform efficiently and adequately for the data distribution process.

A PF tree can potentially range from linear to exponential in growth. However, it is expected and empirical evidence suggests that the average tree size is far from the worst case. As a matter of fact the actual tree size is much closer to the actual number of fragments being generated. This is to be expected since the number of fragments is based on data usage and usage information is used to generate the growth in the PF trees.

The worst case scenario for tree growth is an almost complete binary tree with a depth equal to the number of predicates issued (Figure 5.1a). Such a tree grows exponentially as predicates are inserted and where the current selection consists of all the entity instances of the entity set associated with that tree[14]. The best case scenario is a linear tree containing the number of nodes equal to twice the number of predicates issued[15] (Figure 5.1b). Such a tree grows linearly when predicates are inserted into the tree through consecutive reselection operations (*ie*. the primitive predicate identified by the left-most leaf node comprises the current selection in each consecutive operation).

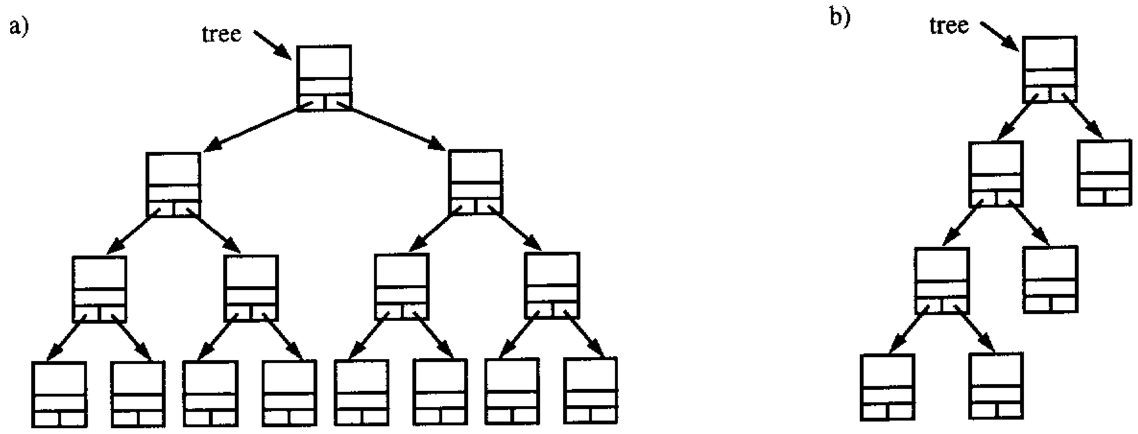


Figure 5.1 The two extremes of PF tree growth: a) exponential b) linear.

The type of selection operation, being either a *reselect*, *aselect* or *nselect*, will determine the active status of resulting primitive predicates and the active status will determine whether or not the leaf nodes of the PF tree will be expanded during a subsequent selection operation (Section 4.3.1). The *nselect* has the least effect on tree growth while the *aselect* operator has potentially the greatest effect. The *aselect* operation may add more primitive predicates to the current selection, thereby increasing the potential for additional predicates to be inserted.[16] Processing a *new* user session also has the potential of increasing tree growth since at the beginning of each new session all primitive predicates are made active.

A PF model was constructed for each individual user session to compare the user sessions and the trees generated from them (see Appendix B for a description of the user sessions and Section 3.4.5 for the tree generation operations). For each user session the number of primitive predicates and nodes generated and the maximum depth of a PF tree were

[14] The actual number of nodes is calculated as $2^{(d-1)}-1$ where d=depth of tree (root node is at d=0).
[15] This is excluding the root node which is associated with all the instances of an entity set. The actual number of nodes, including the root node, is calculated as $2n+1$ where n= number of primitive predicates.
[16] Recall from Section 3.3.1 that new predicates are obtained from the *reselect* and *aselect* operations and inserted only under active nodes.

accumulated across all 56 entities in the GEOMINE database. The results are provided in Table 5.1. Note that the figures for the PF trees do not include any pruning.

Table 5.1: Summary of GEOMINE user sessions and associated PF model

GEOMINE user session	Total number of operations	Number of selection operations	Number of user predicates	Number of primitive predicates for all PF trees#	Number of nodes for all PF trees#	Maximum depth of PF tree
1	38	19	10	61	66	5
2	325	32	19	69	82	5
3	889	93	38	152	248	9
4	1133	115	60	171	286	10
5	161	25	14	90	124	6
6	41	19	11	137	218	9
7	46	11	9	62	68	3
8	33	11	9	62	68	3
9	358	169	104	148	240	15

The number of primitive predicates and nodes includes the root node maintained for each of the 56 entity sets in the GEOMINE database.

The application user sessions vary in size ranging from 33 to 1133 operations being executed per session. The *number of selection operations* refers to all selection operations executed within the user session. The number of user predicates was obtained from all the *reselect* operations and from the *aselect* operations that contain a predicate. From the table it is evident that the number of selection operations constitutes from 9.8% (for GEOMINE user session 2) to 50% (for user session 1) of the total number of operations and that there is very little correlation between these percentages and session size (R^2=0.36). Similarly, user predicates are found in anywhere from 40.9% (for user session 3) to 81.8% (for user sessions 7 and 8) of selection operations indicating very little correlation between these percentages and the number of selection operations (R^2=0.16). It is obvious that the number of selection operations and user predicates can vary widely within a user session. More important is the number of primitive predicates which, as explained in Section 2.5.2, are derived from the user predicates and are used to build the PF trees.

A PF model was created for each individual user session and the number of primitive predicates and nodes accumulated across the PF trees for all the entity sets in the database. The number of primitive predicates generated is dependent, not only on the number of user predicates, but also on the order of the selection operations. As already discussed above, the order of the selection operations determines the number of active primitive predicates which in turn directly affects the number of primitive predicates created as new user predicates are inserted in the tree. Note that the number of nodes is directly related to the number of primitive predicates.[17]

As an example, consider the tree sizes for user sessions 3 and 9. Although user session 9 contains more user predicates than user session 3 the number of primitive predicates generated is much less. This is because user session 9 involves a sequence of *reselect* operations for the same entity whereas user session 3 involves more *aselect* operations. The resulting size of the PF tree for user session 9 is closer to linear than that for user session 3. The tree depth provides an indication of the tree shape (*ie.* tending towards exponential or linear growth). In Figure 5.2 the depth of the PF tree with the maximum depth in each user session is plotted versus the number of primitive predicates in that tree[18]. This is compared to the two extreme tree sizes of linear and exponential.[19]

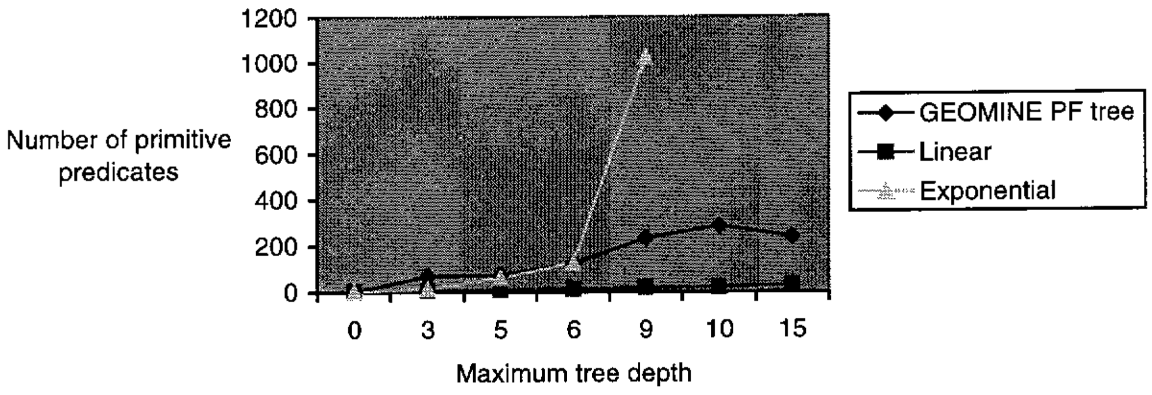


Figure 5.2 Comparison of PF tree sizes for the maximum depth tree in each user session

A PF model was also constructed for nine user sessions containing operations randomised from the original GEOMINE user sessions. The results are provided in Table 5.2. These

[17] For *n* primitive predicates (*ie.* leaf nodes) in a PF tree the number of nodes can be calculated as $2n-1$. Note that the figures in the Table 5.1 refer to all 56 PF trees.
[18] Where two or more user sessions contained PF trees with the same maximum depth, the number of primitive predicates is calculated as the average of those trees.

user sessions are much closer in size[20] (*ie*. number of operations) to each other than the original user sessions simply because the operations were distributed quite evenly among the different user sessions. The number of user predicates still varies among the user sessions but the range of variation has been reduced (17-54) relative to the original user sessions (9-104). Both sets of data are representative of typical GEOMINE user sessions. As explained in Section 4.2.2 empirical evidence based on the GEOMINE application suggests that a typical user session size would consist of under 350 commands consisting of 30 user predicates in 60 selection operations.

Table 5.2: Summary of randomised user sessions obtained for the GEOMINE application

GEOMINE randomised test session	Total number of operations	Number of selection operations	Number of user predicates	Number of primitive predicates for all PF trees#	Number of nodes for all PF trees#	Maximum depth of PF tree
T1	401	52	24	73	90	4
T2	401	42	21	68	80	4
T3	180	70	42	106	156	8
T4	402	52	23	97	138	6
T5	432	37	17	84	112	5
T6	213	88	54	133	210	9
T7	358	38	19	77	98	5
T8	429	38	23	85	114	7
T9	208	77	51	110	164	7

The number of primitive predicates and nodes includes the root node maintained for each of the 56 entity sets in the GEOMINE database.

The PF trees constructed for the original and randomised user sessions shown above are not pruned at all. Some of the fragments identified from the resulting PF trees may not be valid because they are empty. As explained in Section 3.3.2 pruning is an important step for both identifying valid fragments and restricting excessive tree growth. The bar charts in Figure

[19] The number of primitive predicates for a linear tree is calculated as $n+1$ and for an exponential tree as 2^n where n is the tree depth. Note that the root node is at a depth of zero.
[20] Note that the randomised user sessions are not *equal* in size. The randomisation is based on all the GIS commands including those for "house-keeping" purposes. The table shows only operations that execute on the data.

5.3 show the effects of pruning for the original and randomised user sessions of GEOMINE. The pruning is performed on primitive predicates that define empty fragments.

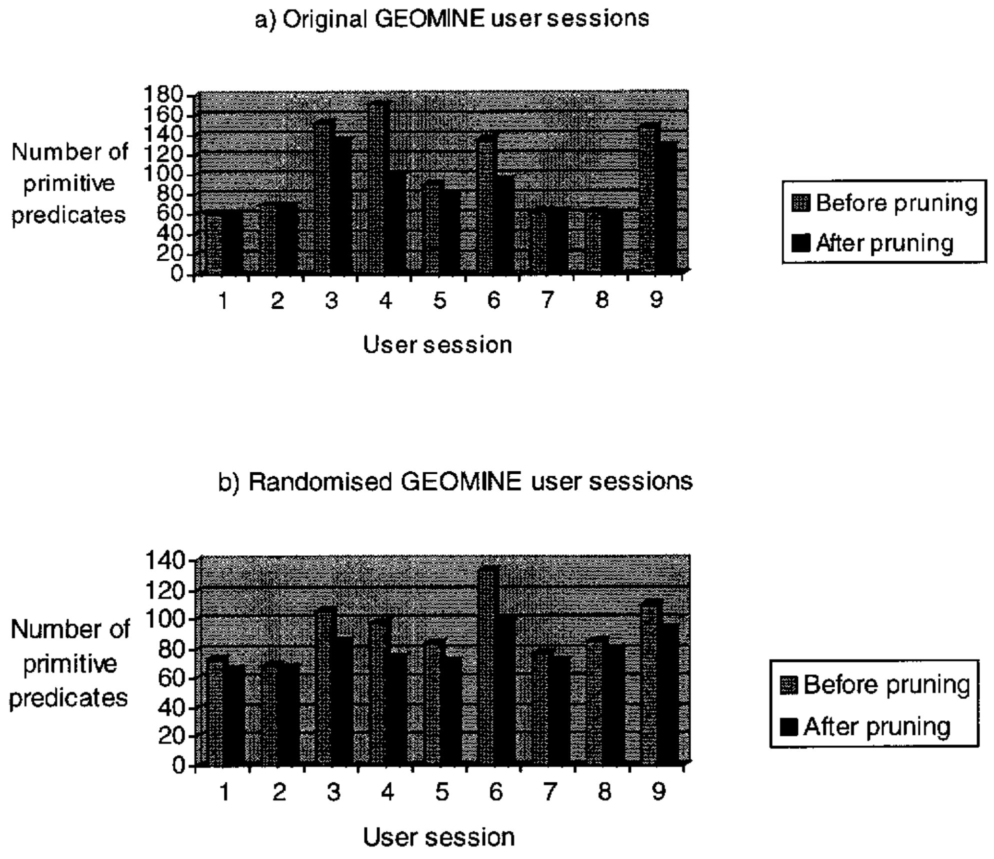


Figure 5.3 Effect of pruning while processing a) original and b) randomised GEOMINE user sessions

Note particularly for large PF trees how pruning causes a marked reduction in the number of primitive predicates. This is expected as explained in Section 4.2.2 since many of the fragments produced from user predicates would not overlap with those of other user predicates and therefore the resulting empty fragments can be pruned. The average number of primitive predicates per user session dropped, as a result of pruning, from 105.8 to 87.7 for the original set of data and from 92.5 to 78.3 for the randomised set of data. These results show a similar reduction in primitive predicates across the two data sets, the reduction being 17.1% for the original data and 15.4% for the randomised data. A comparison between the original and randomised data also reveals that, as expected, randomising the order of insertion of predicates results in no noticeable increase in the number of primitive predicates in the PF tree (see Section 3.3).

The improvements indicated here are based purely on an average across the user sessions which, in Figure 5.3, are dealt with independently from each other. Of course, consecutive user sessions are very much interdependent, often referring to the same entities and the same user predicates (Coleman *et al.* 1992). This is explored in the following sections where they are collectively considered in the distribution methodology.

5.3 Evaluation of Geographic Data Distribution Methodology

The geographic data distribution methodology involves a number of stages as explained in Section 4.2. The distribution requirements analysis and fragmentation design stages result in a fragmentation of a geographic database based on user applications. Such a fragmentation is evaluated in Section 5.3.1. The partitioning and allocation strategies involving the aggregation of these fragments and the assignment of the resulting partitions to computer network sites are evaluated in Section 5.3.2. The final part of the methodology involving data migration is then evaluated in Section 5.3.3.

5.3.1 Evaluation of fragmentation

User sessions cannot simply be treated individually and handled in isolation from each other, but are collectively handled in the data distribution methodology which more closely reflects their occurrence within an application. According to the methodology explained in Section 4.2.2 the PF trees are constructed by processing each user session consecutively and accumulating any new primitive predicates identified in the user sessions. Empty fragments are then pruned after processing each individual user session. Figure 5.4 shows the marked effect of pruning the primitive predicates as they are accumulated for both the original and randomised GEOMINE user sessions.

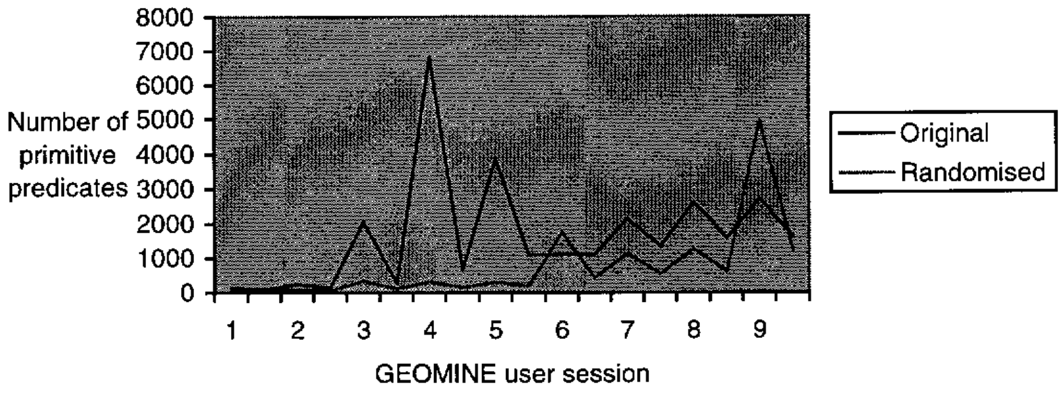
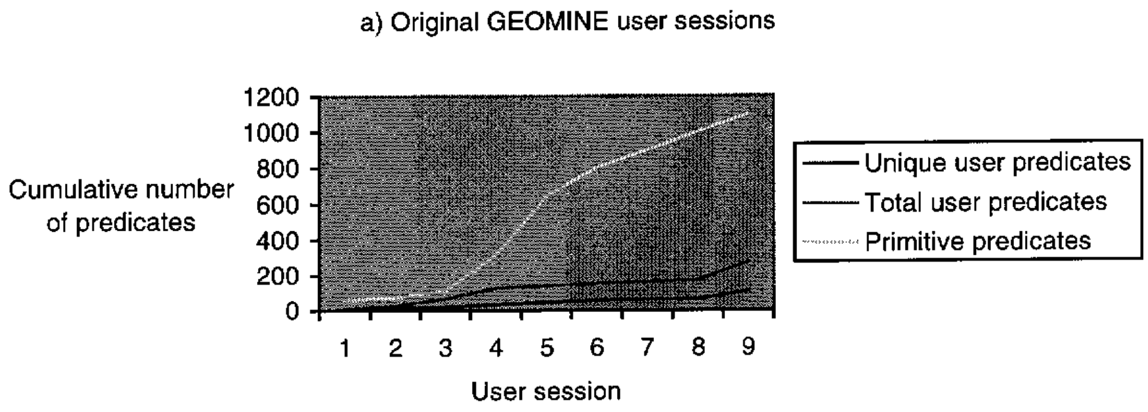


Figure 5.4 Effect of pruning for consecutive processing of application user sessions

The addition of predicates from the user sessions can result in substantial tree growth as appears evident, for example, for the original user sessions 4 and 5, and the randomised user sessions 6 and 9. These user sessions involve a greater number of queries specifying user predicates that identify data not already referenced by previous user sessions. For example, the original user sessions 4 and 5 access a range of topographic and mineral occurrence data across many different regions not previously accessed (see Appendix B). This data is identified with new predicates that are inserted in the tree and result in tree growth and an increase in primitive predicates. However, many of these primitive predicates relate to empty fragments and are therefore pruned. Only those fragments containing the data from the regions accessed during the user session are nonempty. As expected (see comments in Section 5.2) pruning has a dramatic effect in reducing tree size and is therefore an important step in limiting tree growth.



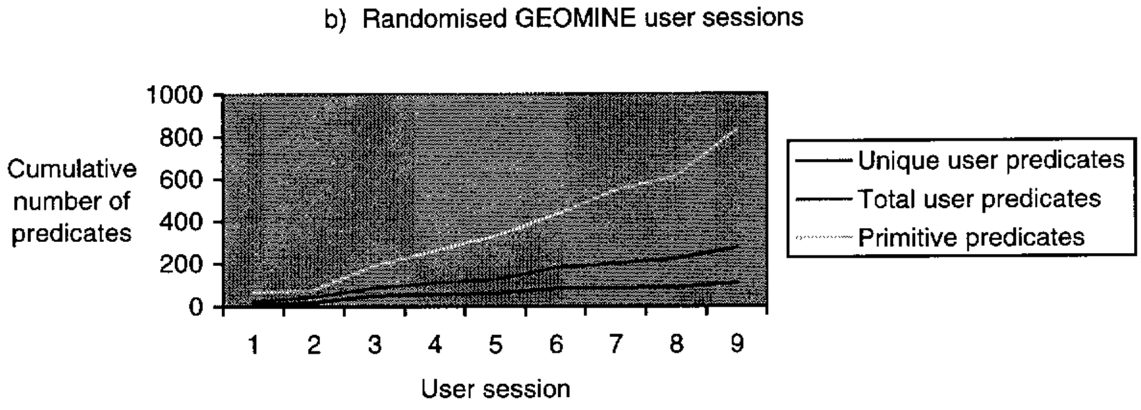


Figure 5.5 Comparison of cumulative numbers of user and primitive predicates for a) original and b) randomised user sessions

The number of primitive predicates is related to the number of unique user predicates obtained from the user sessions. Figure 5.5 displays plots of the cumulative number of user and primitive predicates for both the original and randomised GEOMINE user sessions. Both the *total* number and the number of *unique* user predicates are displayed.

The plots show that even though the number of primitive predicates increases with the increase in unique user predicates, the growth is not exponential but closer to linear. This is expected since the number of fragments, identified by a PF tree, should be proportional to the number of accesses to an entity set by an application.

The fragmentation index identified in Section 4.3 provides an indication of how fragmented a database is. Figure 5.6 shows how the fragmentation index is affected as the user sessions are processed. As expected, the fragmentation increases as the number of user sessions are added. The number of fragments generated is dependent on the overlap among fragments identified by unique user predicates. As more predicates are obtained from the applications, further overlap occurs resulting in additional fragments. The reduction in the number of fragments, as shown at the end of each graph line in Figure 5.6, is due to the pruning of predicates whose fragments are to be allocated to the same site. This again shows the importance of the effect of pruning in the fragmentation process.

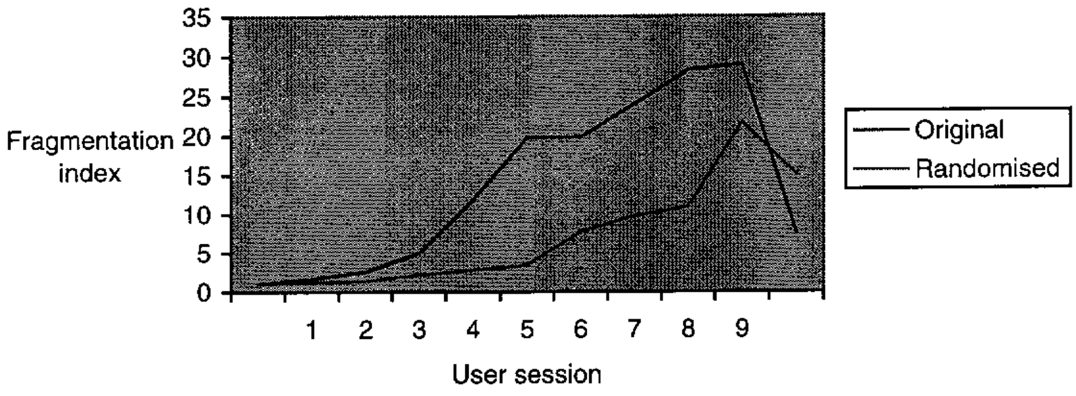


Figure 5.6 Fragmentation resulting from GEOMINE user sessions

The amount of overlap relative to the number of user fragments identified from the applications is measured using the overlap index as described in Section 4.3. The user fragments are identified by the number of unique user predicates obtained from the user sessions of an application. The overlap index is plotted in Figure 5.7 for both the original and randomised user sessions where the user sessions were processed consecutively. Similar to the previous figure, the reduction in the index at the end of the graph indicates the effects of pruning fragments that are to be allocated to the same site.

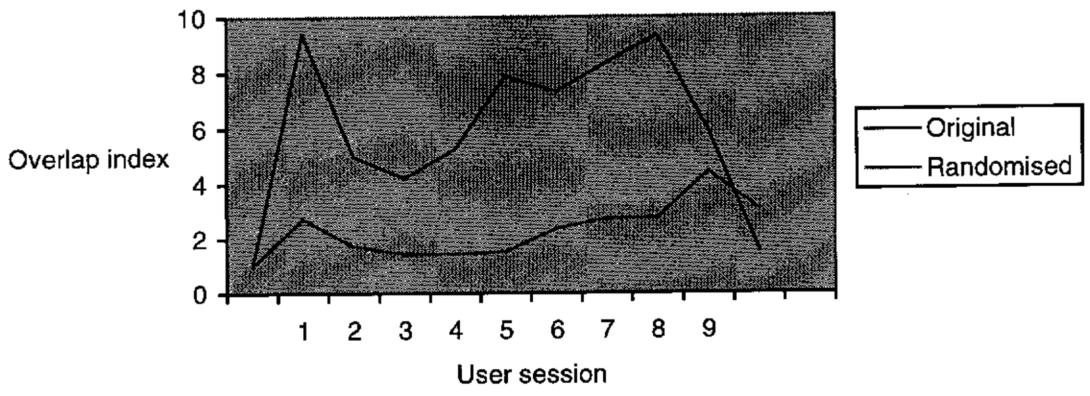


Figure 5.7 Overlap between user fragments identified from GEOMINE user sessions

The overlap between user fragments remains fairly constant as more unique user predicates are identified from the application. Even though there is an increase in the number of primitive predicates as more fragments are generated, the increase is proportional to the number of user predicates obtained from the applications. Because the overlap index is relatively stable, the number of fragments is predictable and the number of primitive predicates in the PF model can be maintained at a manageable level.

Figure 5.7 indicates that the overlap index for the randomised user sessions is less than that for the original user sessions. This is due to the randomisation of the user queries where there is a greater potential of a sequence of queries resulting in no data being returned. For the original user sessions, the user would not normally continue querying from a current selection producing no data. With the randomised user sessions, further queries may result on such a selection producing empty fragments that require pruning. The result is less fragments overall (see also Figure 5.4) and a smaller overlap index.

5.3.2 Evaluation of partitioning and allocation strategy

After the user sessions are processed a data distribution strategy can be generated from the fragmentation obtained from the PF model. Before the actual partitioning and allocation is carried out further pruning may take place for predicates that are small, infrequently accessed or not recently accessed, as discussed in Section 3.3.2. Such pruning ensures that predicates and fragments that are not relevant are discarded. The remaining fragments are then allocated to sites using the *site frequency* allocation method and a data migration occurs according to the resulting distribution strategy (Sections 4.2.3.2 and 4.2.4).

As explained in Section 4.5 the testing strategy used to evaluate the data distribution methodology uses a 3-fold cross-validation approach. The data used for the cross-validation approach was taken from both the original and randomised GEOMINE user sessions. The original GEOMINE user sessions were divided into three groups: user sessions 1-3 in the first group, 4-6 in the second group and 7-9 in the final group[21]. The same thing was done for the nine randomised user sessions. Then for each test run, two groups are used as the training data and one group as the validation data. Table 5.3 lists the test runs.

In order to compare the methodology the training set of data was allocated using three different methods:

1. All data allocated to site 4 - referred to as *one-site* allocation method.[22] The three validation data user sessions were then allocated to sites 1, 2 and 3 respectively. This represents the worst-case scenario where all the data will have to be transferred to another site at which it is used.

2. Data randomly allocated to sites.

3. Site frequency allocation method used in the methodology.

Table 5.3 List of test runs using cross-validation approach

Test Run	***Data set***	***Training set***	***Validation set***
1	Original GEOMINE	1-3, 4-6	7-9
2	Original GEOMINE	1-3, 7-9	4-6
3	Original GEOMINE	4-6, 7-9	1-3
4	Randomised GEOMINE	1-3, 4-6	7-9
5	Randomised GEOMINE	1-3, 7-9	4-6
6	Randomised GEOMINE	4-6, 7-9	1-3

[21] It is logical to group the user sessions in the order in which they appear in the application since consecutive user sessions are likely to contain more similarity than non-consecutive ones in the manner in which they access data.

For all these allocation methods, the three user sessions of the validation set were allocated to three separate sites arbitrarily chosen. A distribution strategy using the site frequency allocation method was then obtained for the validation set of data. The distributions of the training set and validation set were then compared by determining the volume of data that must be transferred and processed for the validation set given the site allocations for the training set of data. The smaller the amount of data transferred the better the data distribution performed for the validation set. By using different allocation methods for the training set, the performance of the site frequency method could be compared relative to the one-site and random allocation methods. The results[23] are displayed in Figure 5.8.

The one-site allocation, as expected, is *always* worse than the site-frequency method since all the data must be transferred to a new site to be used. The site-frequency method assumes that some of the data, allocated based on the training set, is already correctly placed for the validation set. The random allocation method performs well when the resulting data allocation happens to be close to the data allocation for the validation set.

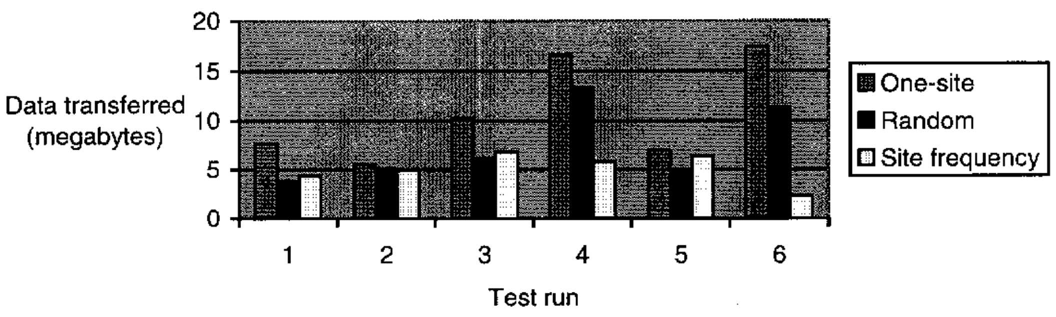


Figure 5.8 Cross-validation results for testing data allocations

The results indicate that in all test cases the site frequency allocation method performs better than the one-site allocation method. It also performs as well as, and in many cases better than, the random allocation method. For test run 3, the random allocation method actually performs better than that of site frequency. This is because the validation set involving the original GEOMINE users sessions 1, 2 and 3 access different data than the training set. Because the training set is a poor indication of data access for this particular validation set,

[22] Site 4 was arbitrarily chosen. It does not matter which site is chosen as long as *none* of the data in the validation set is allocated to this particular site.
[23] Note that the results for test runs 4 and 6 were scaled *down* by 15 and 8 respectively for the purposes of comparison with the other runs.

the site frequency allocation doesn't improve the access and a random allocation happens to perform better.

Test run 5 has a similar result where the random allocation performs better. To determine if this was due to different access patterns between the training and validation sets, the user sessions in the validation set were randomised to be assigned to different computer network sites. The result was that the site frequency allocation method performed better than the random allocation method indicating that the access of the former method happened to be a better match for the placement resulting from the training set.

One further aspect to note from Figure 5.8 is that the last three test runs, which refer to the randomised user sessions, generally involve more data movement between training and validation sets. This is due to the randomisation of the user predicates among the user sessions which therefore contain more variation in the predicates and hence more data being accessed within each individual session.

5.3.3 Evaluation of dynamic data migration strategy

Once a data placement has been obtained and the data has been migrated to the new sites, a dynamic data migration strategy will perform data migration on an ongoing basis. As explained in Section 4.2.4 such a strategy involves the periodic migration of data after the application information has been extracted from multiple user sessions. It was evaluated using all eighteen original and randomised user sessions. The sessions were divided into sets for which a data migration operation was issued. Three different test cases were constructed using different sets of user sessions as shown in Table 5.3. A *k*-fold test case refers to *k* sets of user sessions.

For each test case a data placement was determined for the first test set of user sessions and a data migration initiated. Then the second test set of user sessions was processed by GEODDIS and a new data placement obtained. A further data migration was issued and the process repeated for all test sets.

In addition to pruning empty/small fragments and fragments destined for the same site, pruning was carried out at the end of each test set of user sessions to remove predicates that were not recently accessed. In particular, predicates accessed in a previous test set and not in the current test set were pruned.

Table 5.4 List of test cases constructed for evaluating dynamic data migration for the GEOMINE application

Test set of user sessions	***2-fold test case***	***3-fold test case***	***6-fold test case***
1	Original: 1-9	Original: 1-6	Original: 1-3
2	Randomised: 1-9	Original: 7-9 Randomised: 1-3	Original: 4-6
3	-	Randomised: 4-9	Original: 7-9
4	-	-	Randomised: 1-3
5	-	-	Randomised: 4-6
6	-	-	Randomised: 7-9

Figure 5.9 plots the number of primitive predicates maintained in the PF trees during data migration for each of the three test cases identified in Table 5.4. The test sets of user sessions are processed in the order shown in the table.

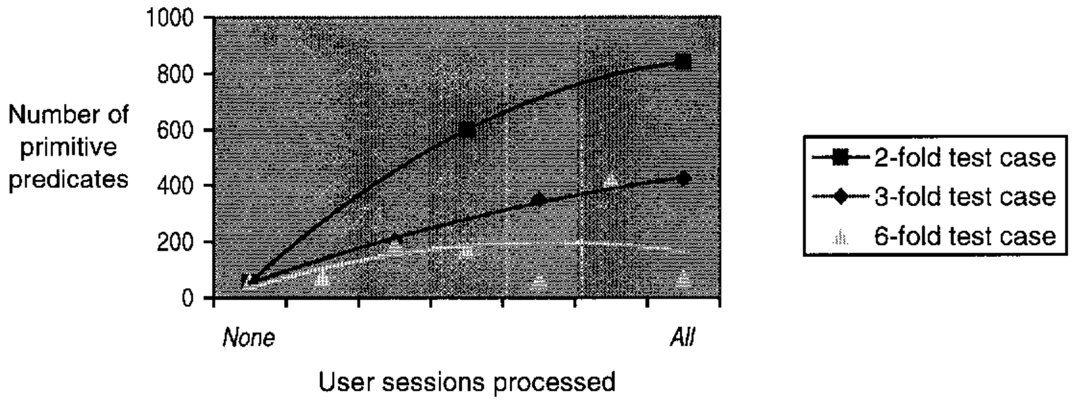


Figure 5.9 Dynamic data migration for three test cases of Table 5.4

The result shows a stabilising effect of the number of primitive predicates and hence of PF tree sizes. This is due to the fact that only the most recent "history" of data usage is maintained in the PF trees. Pruning of predicates is an important part of the process in controlling tree growth and ensuring that the fragmentation identified by the PF trees reflect the most recent usage of data by the applications.

Because data migration occurs more frequently in the 6-fold test case the variation among the user sessions is more apparent than for the 2-fold or 3-fold cases, as is evident from Figure 5.9. In addition, the total number of primitive predicates identified is less since fewer user sessions make up the "history" of data usage. The advantage of more frequent migration is then that the data distribution more closely reflects the data usage in each user session and the PF tree sizes are reduced. However the cost of migration would be greater as explained in Section 4.2.4. Further, even though fewer primitive predicates are generated, there is a greater fluctuation where predicates, previously inserted and then pruned from the PF tree, are used again by future (not necessarily consecutive) user sessions and have to be reinserted. As indicated in Section 4.2.4, the DBA must use his/her knowledge or prediction of data usage to determine the frequency of migration. Further research is necessary to obtain (or predict) and use higher-level knowledge of data usage patterns among current (and future) user sessions to establish a migration frequency which reflects the current application environment.

5.4 Conclusions

The results outlined in this chapter indicate that the PF model, described in Chapter 3, can effectively be used in the geographic data distribution methodology detailed in Chapter 4.

The first part of the problem as outlined in Section 2.7 was to develop and implement a model to store, access and maintain information to be used for distribution. The PF model uses PF trees to provide an elegant means of accumulating geographic data usage information and translating user predicates to primitive predicates. From the implementation in GEODDIS using the GEOMINE application, it is apparent that pruning techniques are effective in limiting the size and shape of a PF tree and thereby controlling the number of primitive predicates obtained. Comparing the results with a randomised version of the GEOMINE user sessions indicates that changing the order of the selection operations and user predicates has no apparent effect on the PF tree growth. The overlap index indicates how the fragmentation is proportional to the number of user predicates and hence is predictable and manageable.

The final part of the problem was to develop and test a methodology incorporating the PF tree to obtain a distribution for geographic data. The fragmentation filter process effectively uses the pruning techniques on the PF tree to only maintain fragments that are relevant. The result is that tree growth is not exponential but closer to linear as demonstrated by both the original and randomised GEOMINE sessions. The cross-validation tests revealed that the site frequency allocation method performs well for applications with similar data usage patterns to those on which the allocation and placement were based; otherwise it performs more like a random allocation. In either case, the allocation was much more efficient than having all the data allocated to one separate site.

Data allocation is dependent on how the data is partitioned and conversely partitioning is dependent on how the data will be allocated to network sites. While some methodologies involve partitioning followed by an allocation stage (Ceri and Pernici 1985) the PF model facilitates the simultaneous creation of partitions and their allocation to sites.

Dynamic data migration of the GEOMINE data for a series of test cases indicates that the number of fragments generated by the distribution stabilises over a period of time. In general, the peak number of fragments generated increases when the interval of migration is increased. Once again, pruning is used to remove unnecessary predicates and to assist in restricting tree growth. The DBA must be involved in determining the frequency of migration that is dependent on the pattern of data usage among user sessions and on the cost of performing a migration.

The geographic data distribution methodology detailed in this research is a great improvement over the mostly-manual methods outlined in the literature and certainly over the static and *ad hoc* methods commonly used for data placement. It is also very necessary for GIS to take advantage of modern distributed data environments so that data can be efficiently shared and accessed by the many and varied applications.

6 Summary and Conclusions

The background of the data distribution design problem is detailed in Chapter 2. The literature deals primarily with the file allocation and the fragmentation problems with little work being done to incorporate geographic information. The fragmentation strategies are extended to include spatial data and a query-adaptive solution is sought which integrates the fragmentation, allocation and data migration solutions. Most solutions identified in the literature deal with these problems independently of each other.

Chapter Three details the Predicate Fragmentation model. Insertion and deletion operations are defined to add and prune predicates that are stored in a PF tree. The PF model provides an elegant and efficient means of storing and maintaining predicates and usage information obtained from user applications. Primitive predicates, identified from the PF trees, identify the fragments that are used further in the data distribution process.

The methodology for geographic data distribution uses the PF model and is explained in Chapter Four. Database, application and site information is obtained in the distribution requirements stage and identifies the user predicate and usage information important for the following stages. For the fragmentation stage, the PF trees are constructed and maintained with this information. A fragmentation filter process involving pruning methods is used to identify the fragments that are valid. The site access allocation method is based on the fragment usage information and determines how the fragments are assembled into partitions and assigned to computer network sites. Dynamic data migration is important to maintain the currency of the placement.

The methodology was implemented in GEODDIS and the results and evaluation are presented in Chapter Five. Results using the GEOMINE application indicate that PF tree growth is closer to linear than it is to exponential. It is particularly the pruning process that is important to restrict tree growth. The fragmentation index and overlap index indicate that the number of fragments is related proportionally to the number of user predicates identified from the applications.

The cross-validation tests revealed that the site frequency allocation method performs well when applications have similar data usage patterns. For very dissimilar applications, the performance is similar to a random allocation. The dynamic data migration strategy, involving the pruning of predicates which relate to previous data placements, ensures that the PF trees and the data placement is current. Results show that the fragmentation becomes relatively stable with fluctuations related to the different data usage patterns between migration intervals. The DBA must maintain a role in determining the frequency of data migration.

The contribution of this research to the geographic data distribution problem can be summarised with respect to the objectives identified in Section 1.5 (and further detailed in Section 2.7):

1. Geographic data distribution is affected by the data usage of applications. Because high-level factors are difficult to identify and quantify, low-level usage information identified from application logs are obtained for the distribution process.

2. The primitive predicate identified from a PF tree is the basic unit of fragmentation. The PF tree provides a suitable data structure for maintaining user predicates and associated usage information so that these basic unit fragments can be identified.

3. The methodology developed integrates the fragmentation, allocation and migration stages in the geographic data distribution process. The methodology uses usage information to determine how the data is decomposed into fragments, assembled into partitions, allocated to computer network sites, and redistributed to reflect a dynamic application environment.

4. The implementation of the methodology in GEODDIS and its testing with the GEOMINE application indicates that it is effective in determining a geographic data distribution. It incorporates the PF model efficiently and performs well particularly when successive applications share similar data access patterns.

6.1 Future Research Directions

This research presents only a small beginning of the further integration of GIS and distributed data systems. As GIS continue to evolve, further work is required to effectively develop them to incorporate distributed technologies. A number of issues to be considered in further research can be identified:

- The logging facilities of GIS need to be expanded to include query cost and performance information. The information provided by the logs should be harmonised with the information necessary in the distribution requirements stage. Better distribution cost information can then also be obtained and utilised.

- Data replication strategies need to be explored and developed for geographic data and incorporated into the distribution design problem. These must be integrated along with the fragmentation, allocation and migration strategies.

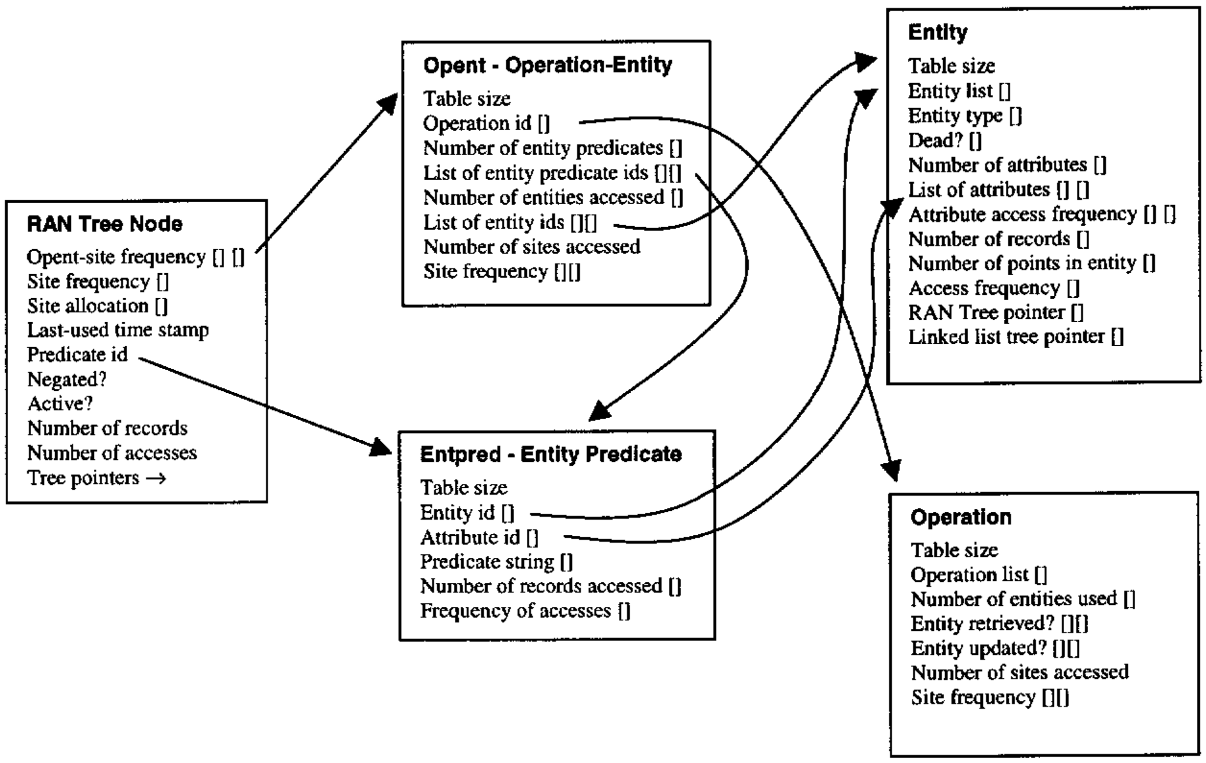
- GIS need to more efficiently implement and optimise queries and take advantage of fragmentation strategies for spatial and attribute data. In particular, vertical fragmentation and spatial fragmentation (both with and without stored topology) need to be more effectively integrated into GIS queries.

- The interval of data migration needs to be further explored so that fully automated methods can be employed. Further information on data usage for, not just individual user sessions, but groups of sessions may need to be examined to determine appropriate intervals of migration.

- Higher level usage information must be examined so that better decisions regarding distribution and migration of geographic data can be made. This may involve information on types of applications, user profiles, application project stages, and organisational policy. Some of these issues have been identified by Goodchild and Rizzo (1986).

Appendix A - GEODDIS Data structure

The data structure used for the GEODDIS implementation is indicated in the following diagram. Note that the square brackets "[]" are used to indicate the dimensionality of the attribute - one pair of brackets is a one-dimensional array and two pairs of brackets is a two-dimensional table.



Appendix B - GEOMINE User Sessions

The user sessions used in this research for testing the geographic data distribution methodology are listed and briefly described as follows:

GEOMINE User Session 1

This user session accesses the mapsheets of Perth, Geraldton, Port Hedland, Albany and Busselton and displays the map sheet boundaries and the mineral occurrences within them.

GEOMINE User Session 2

Queries are concentrated on the Albany region where developing "bedrock opencut" gold mines are accessed. The area is defined by a rectangular window specified by the user using the mouse. A number of queries regarding secondary minerals such as copper are made over the same region. Other data accessed in this region include coastline, map sheet boundaries, national park boundaries, rivers, lakes, cities and towns, railways, highways and roads

GEOMINE User Session 3

Data accessed in this user session concentrated on one region. Developing and abandoned mines are accessed in the Laverton region. Several selections and queries are made based on the type of mine (*ie.* opencut, underground) and on the mineral type found there (gold being the primary mineral and copper the secondary). In addition a number of other features are selected and accessed including map sheet boundaries, national park boundaries, rivers, lakes, cities and towns, railways, highways and roads. Some of these queries occur on a smaller area within the Albany region. For some queries, the smaller area was selected using the mouse and placing a rectangular window over the area of interest. For other queries, the area was selected by specifying the maximum and minimum coordinates.

GEOMINE User Session 4

The focus of this user session was to display general topographic and mineral occurrence information for a number of regions and to access more specific regions with detailed queries. The regions accessed involved Kalgoorlie, Ningaloo and Sandstone, Glengarry, Edjudina, Lennard River and Laverton. The topographic information selected and displayed included mapsheet boundaries, railways, cities and towns, highways and roads. The latter four regions were accessed in greater detail with queries involving gold and copper deposits and mine types. Smaller areas were identified within these regions and zoomed into. Further queries involved identifying and selecting specific mineral occurrence points and displaying their properties.

GEOMINE User Session 5

The GEOMINE5 user session accesses primarily attribute information for various themes such as coastline, 1:250 000 map sheets, mineral occurrences, and some of the attribute tables containing mine types, mine development status and mineral types. A number of attributes for particular features of mines, map sheets, cities and towns, lakes and highways are then retrieved and updated. Most of the accesses refer to data in the Laverton region.

GEOMINE User Session 6

Some of the mine attributes were first accessed followed by all developing mines and then a particular mine called "Radio Hill". The Edmund region was then selected and mines referred to as "Mosquito Creek" and "Elsie Creek" were then queried. In addition, the rivers in the Edmund region were queried.

GEOMINE User Session 7

Once again data in the Laverton region is selected and accessed and a particular area within the region is selected. Geology is the main theme queried with a number of geology types selected. Other themes used mainly for display include map sheets, national parks, lakes, roads, railways and towns.

GEOMINE User Session 8

The Ravensthorpe region, adjacent to Laverton, is selected and queried. As for the previous user session, most queries involve geology with references to map sheets, national parks, lakes, roads, railways and towns.

GEOMINE User Session 9

Geology, roads and highways, railways, national parks, rivers and developed mines are the themes accessed by this user session. Queries on all this data were executed for a number of regions defined by their map sheets. The names of the map sheets accessed are Laverton, Edjudina, Kalgoorlie, Rason, Leonora, Cundeelee, Kurnalpi, Esperance, Albany, Charnley, Mount Elizabeth, Bremer Bay and Ravensthorpe.

Appendix C - GEODDIS Commands and Syntax

The commands developed in GEODDIS to implement the geographic data distribution methodology are listed and briefly explained below. Each command operates on a particular application database which is specified.

WATCH

use <application> watch <watch_file> <site#>

watch_file	Given ArcInfo watch file containing log of a user session
site#	Site at which user session was processed

Reads in each of the commands from the given ArcInfo *watch* file and identifies the entities that are accessed and the user predicates used to specify selections. The predicates are inserted into the PF tree for the corresponding entity. Any accesses to entities are identified and usage information is updated in the PF trees.

LOG

use <application> log <log_file> <site#>

log_file	Given ArcInfo watch file containing log of a user session
site#	Site at which user session was processed

Reads in each of the commands from the given ArcInfo *log* file and identifies the entities that are accessed and the user predicates used to specify selections. The predicates are inserted into the PF tree for the corresponding entity. Any accesses to entities are identified and usage information is updated in the PF trees.

STATISTICS

use <application> [stat | pfstat | allocstat]

stat	General statistics showing entities and properties
pfstat	Displays predicates and contents of PF trees
allocstat	Displays site allocations for each fragment

Generates statistics showing the fragmentation and allocation identified from the PF trees.

ALLOC

use <application> alloc [sitefreq | locality | site <n> | random[1|2|3]]

sitefreq	Site frequency allocation method
locality	Locality allocation method
site <n>	One-site allocation method where site number is specified
random[1|2|3]	Random allocation where three different randomisations can be chosen

Determines an allocation of fragments from the PF trees based on the given method.

CLEAR

```
use <application> clear { <site#> }
```

site#	Site at which usage information is cleared.

Clears the usage information at the specified site. If no site is specified, then the information is cleared at all sites.

BUILDPRED

```
use <application> buildpred
```

Builds a list of ArcInfo queries using the primitive predicates of the PF trees to define the fragments. Each fragment is to be queried to determine its size.

UPDATE

```
use <application> update
```

Takes as input an ArcInfo watch file which has logged the ArcInfo queries executed from the BUILDPRED operation. The PF trees are updated with fragment sizes.

MIGRATE

```
use <application> migrate [ locality | sitefreq | site | random ]
```

locality I sitefreq I site I random	The allocation method used

A new placement for data migration is determined. The old placement is compared with the new PF model and an ArcInfo file of queries is generated identifying the data that must be moved to a new site. This file can then be executed in ArcInfo to effect the migration. This command also produces a "usage" file which is used to determine the costs (see the TEST command). Finally, the file containing the current placement is updated to reflect the new placement.

PRUNE

```
use <application> prune [ fragsize <#records> I lowaccess <#accesses> I outdate <#minutes>
        I samesite [ locality I sitefreq I site I random ]
```

fragsize <#records>	All fragments with a size less than the given minimum are to be pruned
lowaccess <#accesses>	Specifies that fragments with fewer accesses than that given are to be pruned
outdate <#minutes>	Indicates that predicates that were last accessed beyond the given time (in minutes) are to be pruned

samesite [locality I sitefreq I site I random]	Indicates that fragments allocated to the same site using the given allocation method are to be pruned

Prunes all the PF trees based on the pruning method specified.

TEST

use <application> test

Displays the costs associated with migrating data. The files produced by the migration process are used as input to determining the costs.

Appendix D - UNIX Shell Script Example

The methodology was implemented using a UNIX shell script which calls C language programs and the ArcInfo GIS to carry out the functions of geographic data distribution. The script shown below captures the information from all available GEOMINE user sessions (consecutively numbered to distinguish them), produces a data distribution, and then performs a data migration operation to redistribute the data according to the newly derived data placement. See Appendix C for further detail on each of the GEODDIS commands.

```
#!/bin/csh

#
#       GEODDIS - Geographic Data Distribution System
#       Designed and implemented by B. Veenendaal 1998
#
#       Script to obtain usage information and generate a geographic
#       data distribution and migration operation.
#
#
#       Parameters:
#               app             The application whose logs are being processed
#               outf            The output file for logging the distribution process
#               updatef         The ArcInfo commands for querying the database to obtain
#                                       fragment sizes
#               $app.migrating  The ArcInfo commands to migrate the data for the given application
#

set app='geomine'
set outf='goTP.out'
set updatef='go.update'
date > $outf
use $app clear >> $outf

#
#       Obtain the application watch files and build the PF trees. Prune empty fragments after each
#       user session is processed.
#

@ i = 1
while (-e $app{T}$i.wat)
    echo {$app{T}$i}: >> $outf
    echo Processing {$app{T}$i} ...
    use $app watch $app{T}$i.wat $i >> $outf
    use $app buildpred >> $outf
    $ARCHOME/programs/arc arcplot < $updatef >> $outf
    use $app update >> $outf
    use $app prune fragsize 0 >> $outf
    @ i = $i + 1
end
```

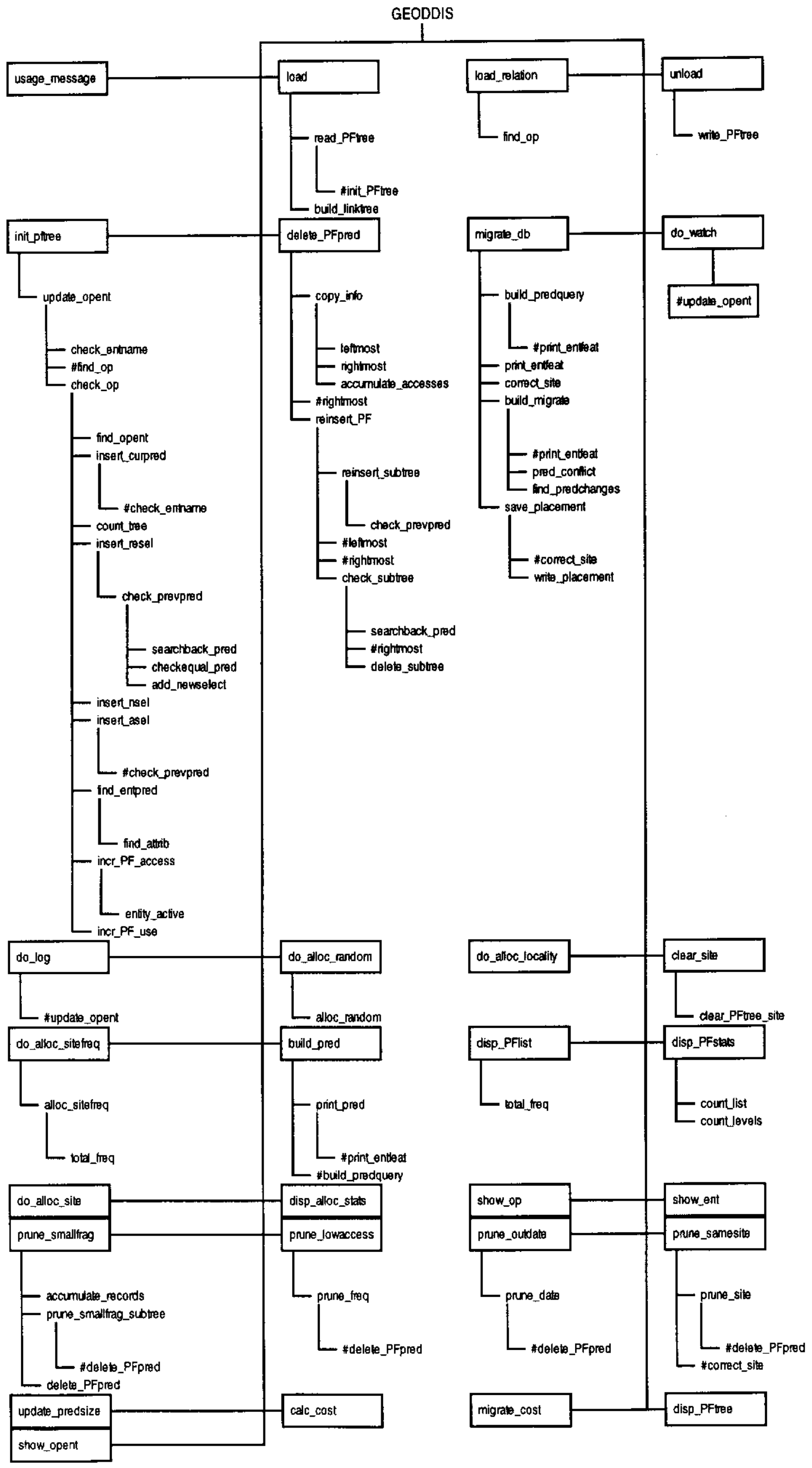
```
#
#       Determine the allocation of fragments to partitions and sites using several
#       different methods. Then prune fragments destined for the same site based on
#       the site frequency allocation method (method 1).
#

use $app alloc sitefreq >> $outf
use $app alloc locality >> $outf
use $app alloc random1 >> $outf
use $app alloc site 1 >> $outf
use $app prune samesite 1 >> $outf
use $app allocstat >> $outf

#
#       Now determine the data placement for a migration, effect
#       the migration in ArcInfo and produce the results showing
#       the migration costs.
#

echo Migrating and testing...
use $app migrate sitefreq >> $outf

$ARCHOME/programs/arc arcplot < $app.migrating >> $outf

use $app test >> $outf
date >> $outf
```

Appendix E - Structure Chart of GEODDIS Modules

The GEODDIS program modules are implemented using C-language. The following *structure chart* lists the modules and their relationship to other modules. The hash ("#") symbol indicates modules that are already expanded elsewhere in the chart. Note that recursive calls to modules are not listed.

Further information on the modules and the associated source code can be obtained from the GEODDIS web page at www.cage.curtin.edu.au/~bertveen/geoddis.html.

GEODDIS Module Structure Chart



References

Abel, D.J., B.C. Ooi, R. Power, K-L Tan, G. Williams, and X. Zhou (1994) The Virtual Database: a Tool for Migration from Legacy LIS. *Proceedings of AURISA'94: the 22nd Annual Internation Conference of the Australiasian Urban and Regional Information Systems Association*, 21-25 November, Sydney, Australia, pp 117-126.

Adam, Nabil R. and Aryya Gangopadhyay (1997) *Database Issues in Geographic Information Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

Aho, Alfred V. and Jeffrey D. Ullman (1992) *Foundations of Computer Science.* Computer Science Press, An imprint of W.H. Freemand and Company, New York.

Alexandria (1996) The Alexandria Project. <alexandria.sdc.ucsb.edu/>, Alexandria Digital Library, September.

Andleigh, P.K. and M.R. Gretzinger (1992) *Distributed Object-Oriented Data-Systems Design.* Prentice Hall, Englewood Cliffs, New Jersey.

Apers, P.M.G. (1982) *Query Processing and Data Allocation in Distributed Database Systems.* PhD thesis, Vrije Universiteit te Amsterdam, The Netherlands.

Batini, Carlo, Stefano Ceri, and Shamkant B. Navathe (1992) *Conceptual Database Design: An Entity-Relationship Approach.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California.

Bell, David and Jane Grimson (1992) *Distributed Database Systems.* International Computer Science Series, Addison-Wesley, Wokingham, England.

Bernstein, P.A. and N. Goodman (1981) Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13:2, pp 185-222.

Bishr, Yaser (1997) Semantic Aspects of Interoperable GIS. ITC Publication Series, CIP-Data Koninklijke Bibliotheek, Den Haag, The Netherlands.

Boffey, Brian (1992) *Distributed Computing: Associated Combinatorial Problems.* Blackwell Scientific Publications, Oxford.

Burrough, P.A. (1986) Principles of Geographical Information Systems for Land Resources Assessment. Oxford University Press, Oxford.

Casey, R.G. (1972) Allocation of cop;ies of a file in an information network. Proceedings AFIPS 1972 Computer Conference, 43, AFIPS Press, Arlington, VA, pp 617-625.

Ceri, S. and S.B. Navathe (1983) A Methodology to the Distribution Design of Databases. *Proceedings IEEE COMPCON Conference*, San Francisco, CA, February.

Ceri, S., S. Navathe, and G. Wiederhold (1983) Distribution Design of Logical Database Schemas. *IEEE Transactions on Software Engineering*, 9:4, July, pp 487-504.

Ceri, S. and G. Pelagatti (1984) *Distributed Databases: Principles and Systems.* McGraw-Hill Book Company, New York.

Ceri, S. and B. Pernici (1985) DATAID-D: Methodology for Distributed Database Design. *Computer-Aided Database Design: The DATAID Project*, A.Albano, V. De Antonelis, and A. Di Leva (Editors), Elsevier Science Publishers B.V., North Holland, pp 157-183.

Ceri, Stefano, Barbara Pernici and Gio Wiederhold (1987) Distributed Database Design Methodologies. *Proceedings of the IEEE*, 75:5, May, pp 533-546.

Chang, S.K. (1972) Database Decomposition in a Hierarchical Computer System. *ACM SIGMOD Proceedings*, pp 48-52.

Chang, Shi-Kuo and Wu-Haung Cheng (1980) A Methodology for Structured Database Decomposition. *IEEE Transactions on Software Engineering*, SE-6:2, March.

Chang, Shi-Kuo and An-Chi Liu (1982) File Allocation in a Distributed Database. *International Journal of Computer and Informatoin Sciences*, 11:5, pp 325-340.

Chen, Peter Pin-Shan (1976) The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1:1, pp 9-36.

Chu, W.W. (1969) Optimal File Allocation in a Multiple Computer System. *IEEE Transactions on Computers*, C-18:10, October.

Chu, Wesley W. and Paul Hurley (1982) Optimal Query Processing for Distributed Database Systems. *IEEE Transactions on Computers*, C-31:9, September, pp 835-850.

Coleman, D.J., P. Morriss and P.R. Zwart (1992) Quantifying GIS Usage in an Organisation: Developing Management Tools. *Proceedings of AURISA '92: Twentieth Annual International Conference of the Australasian Urban and Regional Information Systems Association*, Gold Coast Australia, pp 27-36.

Coleman, D.J., and P.R. Zwart (1992a) Modelling Usage and Telecommunication Performance in Real-time Spatial Information Networks. *Proceedings of the Fifth International Symposium on Spatial Data Handling*, IGU Commission on GIS, August 3-7, Charleston, South Carolina, USA, pp 144-153.

Coleman, D.J., and P.R. Zwart (1992b) Determining GIS Performance Across Broadband Telecommunication Networks. *Proceedings of AURISA '92: Twentieth Annual International Conference of the Australasian Urban and Regional Information Systems Association*, Gold Coast Australia, pp 421-431.

Copeland, G., W. Alexander, E. Boughter, and T. Keller (1988) Data Placement in Bubba. *Proceedings of SIGMOD: International Conference on Management of Data*, Chicago, Illinois, June 1-3.

Cornell, D. and P.S. Yu (1987) A Vertical Partitioning Algorithm for Relational Databases. *Proceedings of the Third International Conference on Data Engineering*, February, pp 30-35.

Cornell, Douglas J. and Philip S. Yu (1990) An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Transactions on Software Engineering*, 16:2, February.

Dangermond, J. (1996) ESRI Technical Directions. Opening speech of *OZRI 10/ESA 5: The 10th Annual Australia and 5th Annual South Asia ESRI and ERDAS User Conference*, Perth, Western Australia, 4-6 September.

Date, C.J. (1982) An Introduction to Database Systems. Third edition, Addison-Wesley Publishing Company, Reading, Massachusetts.

De Antonellis, V. and A. Di Leva (1985) DATAID-1: A Database Design Methodology. *Information Systems*, 10:2, pp 181-195.

DeMers, Michael N. (1997) *Fundamentals of Geographic Information Systems.* John Wiley and Sons, Inc., New York, U.S.A.

Devereaux, D., D. Hudson, and B. Veenendaal (1992) *Geographic Information Systems Concepts.* Curtin University Publication.

Dowdy, L.W. and D.V. Foster (1982) Comparative Models of the File Assignment Problem. *ACM Computing Surveys*, 14:2,pp 287-313, June.

Droge, G. and Schek, H. (1993) Query-Adaptive Data Space Partitioning using Variable-Size Storage Clusters. In *Advances in Spatial Databases*, Third International Symposium SSD'93 proceedings, Springer-Verlag.

Droge, Gisbert (1994) Patchwork: A Query-driven Adaptive Data Space Partitioning. *IGIS'94: Geographic Information Systems*, International Workshop on Advanced Research in Geographic Information Systems, Springer-Verlag, Berlin, pp 214-224.

ESRI (1989) *Librarian: spatial database management for ARC/INFO.* Environmental Systems Research Institute, Redlands, California.

ESRI (1992) *Understanding GIS: The Arc/Info Method.*.Environmental Systems Research Institute, Inc., Redlands, California.

ESRI (1994) *ARC/INFO Data Management: Concepts, data models, database design, and storage.* Environmental Systems Research Institute, Inc., Redlands, California.

Eswaran, K.P. (1974) Placement of Records in a File and File Allocation in a Computer Network. *Information Processing (IFIPS).*

Firns, P.G. (1992) On the Notion of a Spatial Entity in the Context of Data Modelling for Spatial Information Systems. *Proceedings of the Fourth Colloquium of the Spatial Information Research Centre*, University of Otago, Dunedin, New Zealand, May.

Fisher, M.L. and D.S. Hochbaum (1980) Database Location in Computer Networks. *Journal of the Association for Computing Machinery*, 27:4, October, pp 718-735.

Frank, A. (1981) Application of DBMS to Land Information Systems. *Proceedings of the 7th International Symposium on Very Large Databases*, Cannes, France, pp 448-453.

Frank, A. (1988) Requirements for a Database Management System for a GIS. *Photogrammetric Engineering and Remote Sensing*, 54(11), November, pp 1557-1564.

Gavish, Bezalel and Hasan Pirkul (1986) Computer and Database Location in Distributed Computer Systems. *IEEE Transactions on Computers,* C-35:7, July, pp 583-590.

Gavish, Bezalel (1987) Optimization Models for Configuring Distributed Computer Systems. *IEEE Transactions on Computers*, C-36:7, July, pp 773-793.

Gavish, Bezalel and Olivia R. Liu Sheng (1990) Dynamic File Migration in Distributed Computer Systems. *Communications of the ACM*, 33:2, February, pp 177-189.

Goodchild, M.F. (1989) Tiling Large Geographical Databases, in *Design and Implementation of Large Spatial Databases.* Proceedings of the First Symposium SSD'89, Springer-Verlag.

Goodchild, M.F., and B.R. Rizzo (1986) Performance Evaluation and Workload Estimation for Geographic Information Systems. *Proceedings of the Second International Symposium on Spatial Data Handling*, Seattle, Washington, U.S.A., July.

Goyal, P., T.S. Narayanan and F. Sadri (1993) Query Processing in distributed Databases with Nondisjoint Data. *Information Systems*, 18:7, pp 419-427.

Gunther, O. (1988) *Efficient Structures for Geometric Data Management.* Lecture Notes in Computer Science 337, Springer-Verlag, Berlin.

Guttman, A. (1984) R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD'84*, Boston, Massachussets USA, June.

Hevner, A. R. and A. Rao (1988) Distributed Data Allocation Strategies. *Advances in Computers,* Volume 27, Academic Press, Inc.

Hoffer, J.A. and D.G. Severance (1975) The use of Cluster Analysis in Physical Database Design. *Proceedings 1st International Conference on Very Large Databases,* Framingham, Massachusetts.

Hua, K.A., C. Lee and H.C. Young (1993) Data Partitioning for Multicomputer Database Systems: a Cell-based Approach. *Information Systems*, 18:5, pp 329-342.

Hurson, A.R. and M.W. Bright (1991) Multidatabase Systems: An Advanced Concept in Handling Distributed Data. *Advances in Computers,* Vol. 32.

Ingres (1995) Star Architecture. Star User Guide, Chapter 3, CA-OpenIngres DocuROM Library.

Korth, Henry F. and Abraham Silberschatz (1991) *Database System Concepts.* Second edition, McGraw-Hill, Inc., USA.

Kriegel, Hans-Peter, Holger Horn andMichael Schiwietz (1991) The Performance of Object Decomposition Techniques for Spatial Query Processing. *Advances in Spatial Databases*, O Gunther and HJ Schek (Editors), Proceedings of the 2nd Symposium, SSD'91, Zurich, Switzerland, August 28-30, pp 257-276.

Kuspert, K. and E. Rahm (1990) Trends in Distributed and Cooperative Database Management. *Database Systems of the 90s*, A. Blaser (ed), Springer-Verlag, Berlin, pp 263-293.

Laning, L.J. and M.S. Leonard (1983) File Allocation in a Distributed Computer Communication Network. *IEEE Transactions on Computers*, C-32:3, March, pp 232-244.

Laurini, R. and D. Thompson (1992) *Fundamentals of Spatial Information Systems.* Academic Press Limited, London.

Leigh, W.E., and C. Burgess (1987) *Distributed Intelligence: Trade-offs and Decisions for Computer Information Systems.* South-Western Publishing Company, Cincinnati, Ohio.

Lewis, Ted (1996) The Next $10{,}000_2$ Years: Part II. *Computer*, IEEE Computer Society, 29:5, May, pp 78-86.

Li, Ki-Joune and Robert Laurini (1991) The Spatial Locality and a Spatial Indexing Method by Dynamic Clustering in a Hypermap System. *Advances in Spatial Databases*, O Gunther and HJ Schek (Editors), Proceedings of the 2nd Symposium, SSD'91, Zurich, Switzerland, August 28-30, pp 207-223.

McCormick, W.T., P.J. Schweitzer, and T.W. White (1972) Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20:5, September, pp 993-1009.

Mitchell, T.M. (1997) *Machine Learning.* The McGraw-Hill Companies, Inc., United States of America.

Morgan H.L. and K.D. Levin (1977) Optimal Program and Data Location in Computer Networks. *Communications of the ACM*, 20(5): 315-322, May.

Navathe, S.B. and S. Ceri (1985) A Comprehensive Approach to Fragmentation and Allocation of Data in Distributed Databases. Reprinted in J.A. Larson and S. Rahimi (eds.), *Tutorial: Distributed Database Management*, IEEE Computer Society Press.

Navathe, S., S. Ceri, G. Wiederhold, and J. Dou (1984) Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4):680-710, December.

Navathe, Shamkant B. and Minyoung Ra (1989) Vertical Partitioning for Database Design: A Graphical Algorithm. Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon, *SIGMOD Record*, 18:2, June, pp 440-450.

NCGIA (1990) *Introduction to GIS: NCGIA Core Curriculum.* National Center for Geographic Information and Analysis, University of California, Santa Barbara.

Network Wizards (1998) Internet Domain Survey July 1998, <www.nw.com/>, July.

Newton, P.W., P.R. Zwart, and M.E. Cavill (1992) Inhibitors and Facilitators in High Speed Networking of Spatial Information Systems. *Networking Spatial Information Systems*, PW Newton, PR Zwart, and ME Cavill (eds), Belhaven Press, London, pp 251-263.

Noronha, V.T. (1988) A Survey of Hierarchical Partitioning Methods for Vector Images. *Proceedings of the Third International Symposium on Spatial Data Handling*, Sydney, Australia, August.

Oracle (1995a) *Oracle7 Server Distributed Systems: Replicated Data*. Oracle Corporation, Redwood City, CA.

Oracle (1995b) Oracle7 Distributed Database Technology and Symmetric Replication. *White paper*, Oracle Corporation, Redwood Shores, CA, April.

Oracle (1999) Oracle8i Enterprise Edition Partitioning Option: Features Overview. *White paper,* Oracle Corporation, February,<http://www.oracle.com/database/documents/ent_partitioning_fo.pdf>

Orenstein, Jack A. and Frank A. Manola (1988) PROBE Spatial Data Modelling and Query Processing in an Image Database Application. *IEEE Transactions on Software Engineering*, 14(5), May, pp 611-629.

Orlowska, M.E., A.M. Lister, and I. Fogg (1992) Decentralising Spatial Databases. *Networking Spatial Information Systems*, PW Newton, PR Zwart, and ME Cavill (eds), Belhaven Press, London, pp 63-76.

Ozsu, M.T. and P. Valduriez (1991a) Distributed Database Systems: Where Are We Now? *IEEE Computer*, 24(8): 68-78, August.

Ozsu, M.T. and P. Valduriez (1991b) *Principles of Distributed Database Systems*. Prentice-Hall International, Inc., Englewood Cliffs, New Jersey.

Patel, J., J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellmann, J. Kupsch, S. Guo, J. Larson, D. DeWitt and J. Naughton (1997) Building a Scalable Geo-Spatial DBMS: Technology, Implementation and Evaluation. *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 13-15, 26:2, June.

Peuquet, Donna J. (1984) A Conceptual Framework and Comparison of Spatial Data Models. *Cartographica*, 21, pp 66-113.

Ra, Minyoung and Yang-Sun Park (1993) Data Fragmentation and Allocation for PC-Based Distributed Database Design. *Proceedings of the Third International Symposium on Database Systems for Advanced Applications '93*, World Scientific Publishing Co. Pte. Ltd., Singapore, April 6-8 Taejon, Korea, pp 90-96.

Ramamoorthy, C.V. and B.W. Wah (1983) The Isomorphism of Simple File Allocation. *IEEE Transactions on Computers,* C-32:3, October, pp 221-231.

Robinson, J.T. (1981) The K-D-B-tree: A search structure for large multidimensional dynamic indexes. Proceedings of ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, pp 10-18.

Roy, Geoff, Tom Schnugg and Mike McGill (1997) State-of-the-art Network-centric GIS: why and where Internet & Intranet GIS applications are Successful. Proceedings of AURISA 97, Christchurch, New Zealand, November 17-21.

Sacca, Domenico and Gio Wiederhold (1985) Database Partitioning in a Cluster of Processors. *ACM Transactions on Database Systems*, 10:1, March, pp 29-56.

Samet, Hanan (1990) *The Design and Analysis of Spatial Data Structures.* Addison Wesley Publishing Company, Inc., Reading, Massachusetts.

Sandberg, Goldberg, Kleiman, Walsh and Lyon (1985) Design and Implementation of the Sun Network File System. Summary on the Web, accessed November 1998, <das-www.harvard.edu/cs/academics/courses/cs261/readings/sandberg-1985.html>

Satyanarayanan, M. (1993) Distributed File Systems. *Distributed Systems,* S. Mullender (ed), ACM Press, Second edition, pp 353-383.

Sedgewick, Robert (1988) *Algorithms.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, United States of America.

Sheth, A.P. and J.A. Larson (1990) Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys,* 22:3, September.

Silberschatz, Avi, Mike Stonebraker and Jeff Ullman (1995) Database Research: Achievements and Opportunities into the 21st Century. Report of an NSF Workshop on the Future of Database Systems Research, May 26-27.

Sloan, T.M., S. Dowers, B.M. Gittings, R.G. Healey and T.C. Waugh (1992) Exploring GIS Performance Issues. *Proceedings of the Fifth International Symposium on Spatial Data Handling,* IGU Commission on GIS, August 3-7, Charleston, South Carolina, USA, pp 154-165.

Smallworld (1994) Smallworld GIS 2 Customisation Overview. Smallworld document, February.

Sockut, G.H. and R.P. Goldberg (1979) Database Reorganization - Principles and Practice. *Computing Surveys,* Vol. 11, No. 4, December.

Sondheim, M. and R. Menes (1991) General Framework for Geomatics Data Exchange. *GIS Applications in Natural Resources,* M. Heit and A. Shortreid (eds.), GIS World Inc., Fort Collins, Colorado.

Spath, H. (1980) *Cluster Analysis Algorithms: for data reduction and classification of objects.* Ellis Horwood Limited Publishers, Chichester.

Stonebraker, M., P.M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu (1996) Mariposa: a Wide-area Distributed Database System. *The VLDB Journal,* Springer-Verlag, 5:48-63.

Strand, Eric J. (1999) GIS Hardware'99: How Does the Competition Stack Up? *GeoWorld,* Adams Business Media, 12:2, February.

Tewari, Raj and Nabil R. Adam (1992) Distributed File Allocation with Consistency Constraints. *Proceedings of the 12th International Conference on Distributed Computing Systems,* Yokohama, Japan, June 9-12, IEEE Computer Society Press, pp 408-415.

Thomas, Gomer, Glenn R. Thompson, Chin-Wan Chung, Edward Barkmeyer, Fred Carter, Marjorie Templeton, Stephen Fox and Berl Hartman (1990) Heterogeneous Distributed Database Systems for Production Use. *ACM Computing Surveys,* Vol. 22, No. 3, September.

Veenendaal, B. (1990) Sharing GIS Data in a Networked Environment, *Proceedings of AURISA'90: The 18th Annual International Conference of the Australasian Urban and Regional Information Systems Association,* Canberra, Australia, November, pp 151-160.

Veenendaal, B. and D. Hudson (1992) Distributing GIS Data in a Computer Network. *Proceedings of GIS'92 Symposium,* Vancouver, British Columbia, Canada, February.

Veenendaal, B. (1994a) Partitioning Vector Data in a Distributed Geographic Information System. Technical Report 5, Curtin University of Technology.

Veenendaal, B. (1994b) Developing a Model for Geographic Data Distribution in a Distributed Geographic Information System. *Proceedings of the Advanced Geographic Data Modelling Workshop,* Delft, The Netherlands, September.

Veenendaal, B. (1994c) Data Distribution Design for Geographic Information Systems. *Proceedings of AURISA'94: The 22nd Annual International Conference of the Australasian Urban and Regional Information Systems Association,* 21-25 November, Sydney, Australia, pp 615-623.

Veenendaal, B., P. Wilkes and S. Lipple (1995) GEOMINE: Developing a Mineral Database for Western Australia. *Proceedings of AURISA'96: the 23rd Annual International Conference of the Australiasian Urban and Regional Information Systems Association,* Melbourne, Australia, 20-24 November, pp 338-346.

Veenendaal, B. (1997) *Introduction to Geographic Information Systems.* Version 1.1, Course notes, Curtin University.

Vincent, Steve (1995) CyberGIS: Providing Geodata via the World Wide Web. GIS Asia Pacific, Pearson Professional (Singapore) Pte Ltd, Singapore, 1:5, October.

Wah, B.W. (1981) *Data Management on Distributed Databases.* UMI Research Press, Ann Arbor, Michigan.

Wah, B.W. (1984) File Placement on Distributed Computer Systems. *IEEE Computer,* 17:1, January.

Waight, P.J. (1997) Future Direction of Spatial Data Management in a Client/Server Environment. *Proceedings of AURISA 97,* Christchurch, New Zealand, 17-21 November.

Whang, Kyu-Young, Sang-Wook Kim and Gio Wiederhold (1994) Dynamic Maintenance of Data Distribution for Selectivity Estimation. *VLDB Journal,* 3:1, January, pp 29-51.

Wilschut, A. (1994) *Parallel Query Execution in a Main-Memory Database System.* PhD Dissertation, University of Twente, Enschede, The Netherlands.

Wolfson, Ouri and Sushil Jajodia (1992) Distributed Algorithms for Dynamic Replication of Data. Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, San Diego, California, ACM Press, pp 149-157.

Wong, E. and R.H. Katz (1983) Distributing a Database for Parallelism. Proceedings of SIGMOD 83, SIGMOD Record, ACM, Volume 13, Number 4, pp 23-29.

Worboys, Michael F. (1995) *GIS: A Computing Perspective.* Taylor & Francis Ltd., London, UK.

Yu, C.T. and C.C. Chang (1984) Distributed Query Processing. *Computing Surveys*, Vol. 16, No. 4, December.

Yu, Clement T., Man-Keung Siu, K. Lam and C.H. Chen (1984) Adaptive File Partitioning in a Hierarchical Distributed System. Unpublished paper.

Yu, Clement T., Man-Keung Siu, K. Lam and C.H. Chen (1985) Adaptive File Allocation in Star Computer Network. *IEEE Transactions on Software Engineering*, SE-11:9, September, pp 959-965.

Zwart, P.R., and Greener, S. (1994) Some Results and Implications of Performance Tests on Large Nationwide Spatial Databases. *Proceedings of AURISA'94: the 22nd Annual International Conference of the Australasian Urban and Regional Information Systems Association*, 21-25 November, Sydney, Australia, pp 135-142.

Zwart, P.R. and D.J. Coleman (1992) Trade-offs Towards Real-time Spatial Information Networks. *Networking Spatial Information Systems*, PW Newton, PR Zwart, and ME Cavill (eds), Belhaven Press, London, pp 265-281.