Heuristics and Multi-dimensional Physical Database Design

By
Zhongsu Fu

A Thesis Submitted in Conformity with
the Requirements of the Degree
of Doctor of Philosophy

Department of Business Systems Analysis,
City University, London EC1 0HB

October 1991

To my patient husband and my parents, without their love and encouragement this work would never have been completed.

Table of Contents

Appendix A

Figures

Figures in Appendices

Acknowledgement

This work has been done under the support of the department of business systems analysis, the City University.

Many thanks to Dr. Allen Long, who has spent a lot of time in supervising and editing main sections of this thesis.

I am also very grateful to Mr. Ray Long and Mr. Lewis Robert from Cognisys Ltd., who have taken my draft to correct my English in their spare time. The City University library has provided well services for materials contributed to the research, staff are very helpful to assist me to get needed information.

Declaration

No portion of the work referred in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of learning.

Zhongsu Fu

Abstract

An expert system approach has recently been used in parameter selection for VSAM (Virtual Storage Access Method) file organisation [AL87a]. This system has been developed to aid in-house users to apply relevant facts and heuristics to optimise VSAM file design. Multi-dimensional physical database design is more sophisticated and complicated than VSAM file design. The expert system approach can be applied to select and tune physical database design for various applications.

A great deal of work has been done in developing diverse algorithms or access methods to organise automated information on secondary storage devices [FA86b] [FR86] [FR88] [GU84] [HU88a] [KS88a] [KS86] [LO87] [NI84] [OR88b] [OR86] [OT85] [RO81], etc. However, little work has been done to enable designers to select an access method which matches a projected application profile (features and requirements) and perceived strengths and weaknesses of candidate algorithms. This thesis considers a number of grid based algorithms and makes expert assessments of each according to its strengths and weaknesses. It analyses features of various access methods and using expert knowledge matches features for a range of m-d (multi-dimensional) algorithms with corresponding characteristics of an application. The knowledge-based system presented in this thesis can be applied either manually or computerised to give a systematic approach to m-d algorithm selection. A system is proposed to (1) heuristically select an initial algorithm; (2) describe how the selection process is evaluated against actual m-d algorithm performance and (3) show how the results of the evaluation can be used to refine expert knowledge embodied in the selection system. Heuristic assessments are given for several m-d access algorithms. Examples are presented to show how these heuristics are used to select a m-d access algorithm for a specific application. It is reasonable to suppose that the initial heuristic assessments are not entirely accurate. A tuning mechanism for the system heuristics is given in section 4.9. The system selection process is thereby, able to adjust to real world results. Finally, we present a simple example to illustrate how the proposed system works.

Key Words:
M-d Physical Database Design
Expert Systems
Matching
Tuning

" There was a consensus that researchers should build automatic physical database design tools that would choose a physical schema and then monitor the performance of the schema making changes as necessary. This would include adding and dropping indexes, load balancing arm activity across a substantial number of disk arms, etc. Hence, tuning knobs should be removed from the domain of the database administer and manipulated by a system demon. "
- Further direction in DBMS research, M. Stonebraker 1988 -

" It is desirable to have pairs of data item which are required consecutively on some query access path to be physically stored **near** to each other. As the number of queries increases the complexity of arranging for this becomes clear, and so usually only the most prominent queries are privileged to be considered for **optimisation** of their placements. Clearly automated design aids are called for to supplement the human designer's skill and experience. "
- David A Bell, 1987 -

Chapter 1. Introduction

As the size of a computerised database increases, the time taken to access required data item(s) becomes a bottleneck. This bottleneck results from intensive access to secondary storage. A question arises - how, in principle, can data be organised using secondary storage devices so as to allow speedy and efficient access? And how can this organisation be physically implemented? This thesis considers these questions for the case of databases including spatial data, i.e. a m-d (multi-dimensional) data space.

A number of m-d search algorithms have been proposed [FA86b] [FR86] [FR88] [GU84] [HU88a] [KS88a] [KS86] [LO87] [NI84] [OR88] [OR86] [OT85] [RO81], etc. They mainly differ in data space partitioning and the implementation of that partitioning. Partitions fall into two basic types: (1) partitioning a data space in its primary key dimension, dividing a data space in its primary key sequence, 1-d (one dimensional) approach; (2) partitioning a data space in a m-d (multi-dimensional) attribute sequence - grid partitions, dividing a data space to preserve its geometrical proximity, a m-d approach.

An example of the former is the inverted file approach. Data are organised in primary key sequence and are stored near to each other in the primary key dimension. One result is that if the difference between two primary key values (|k1 - k2|) is small then the two data items are likely to be stored in linear proximity. This proximity is preserved solely by the primary key. When other keys are involved in the search of the storage location the proximity may no longer hold. (see Figure 1.1. (a) primary key partition (inverted file)). Here, 2-d (two dimensional) data are stored by the inverted file approach. Points p1, p2, and p3 are data items. Note that along the horizontal axis (primary key dimension) points p1, and p2 are closer than p1 and p3 so that p1 and p2 are likely to be stored physically closer than points p1 and p3. The vertical axis is significant when secondary key searches are required. From the diagram, p1 and p3 are closer on the vertical axes (secondary key) than p1 and p2, but as shown in the diagram, p1 and p2 are stored nearer each other than p1 and p3. The searching efficiency is different for a primary key and a secondary key because of the relative distances along the different axes. This asymmetry is an inherent source of a m-d searching inefficiency in the inverted file partition technique.

An example of the latter is a grid file partition. A data space is divided into grids (see Figure 1.1. (b) grid partition). Points p1, p2, p3 are geometrically close and they are in the same grid cell (R0) so that they are stored together.

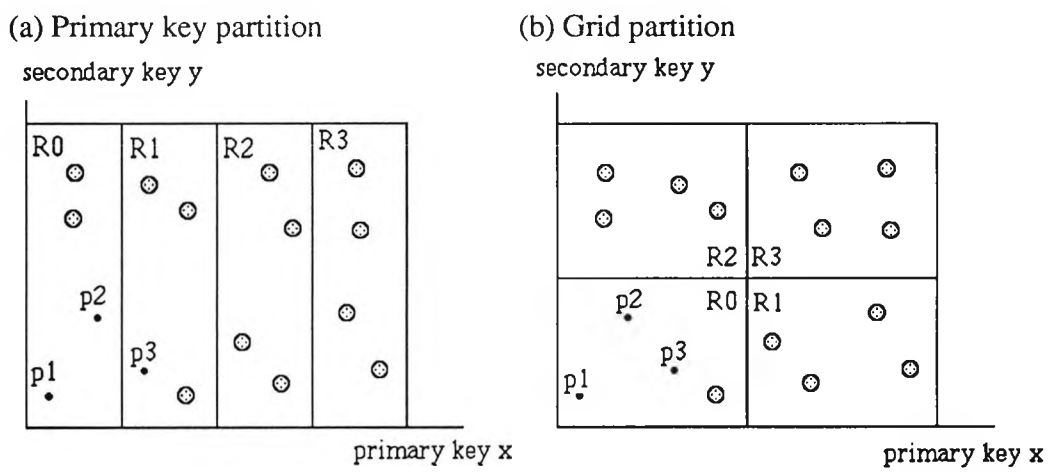


Figure 1.1 Search Space Partition

In the diagram each region Ri for i = 0, 1, 2, 3 corresponds to a data bucket.

An analysis for the inverted file partition and the grid file partition can be found in the Appendix A1. This analysis shows that the complexity of storage and retrieval relates to the query pattern. If in most cases only the primary key is dominant during the life span of a data set, inverted file partitioning can be an advantage. But when other attributes play an important role in the retrieval process, the grid file partition may be more efficient. Grid file partitioning is a better choice for exploring a potentially near-optimal physical database design because the tuning process allows us to adjust the partition depending on various factors including the dominant search attribute(s). Here "dominant attribute(s)" means those attribute(s) that are frequently used for data retrieval. Thus inverted file partitioning can be seen as a special case of the grid partition when the primary key is the dominant attribute because of its high search rate. The grid partitioning approach thus offers more flexibility than the 1-d partitioning approach.

This flexibility of the grid partition, in its various forms [BU83] [FR86] [FR88] [KS88a] [KS88b] [KS86] [LO87] [RU87], allows the possibility of matching the degree of symmetric partitioning with the query pattern for a m-d data space. With inverted file organisation, the data space is always partitioned in one dimension only; whereas with grid file organisation, partitioning can take place in all dimensions (**symmetric partitioning**). Thus grid partitioning aims at geometrical proximity for a spatial search space (a m-d search space) rather than (as with inverted file approach) for a linear search space (a 1-d search space). The grid partition method has been chosen here for physical data organisation because of its flexibility and efficiency in the searching of required data in a m-d search space.

There are various strategies for choosing and implementing grid partitioning, for instance, a scale-based or an interpolation-based grid partitioning [KS88a] and an indexed [FR88] [TA82] or a hashing function implementation [KS88b]. These strategies are application-oriented. If the data distribution is such that the number of data items in each non-empty grid cell is roughly equal then the EXCELL [TA82] scheme, which divides a m-d space into equal sized grid cells, may be used effectively. Note however, that an equal sized grid cell partitioning method can itself be implemented by an indexing approach or by a hashing function. Consequently, further tuning may be required based on the differences between these two implementation techniques.

When a number of empty grid cells are created by a partition, these two methods will differ in terms of storage utilisation and speed. In the case of the indexing implementation, extra index entries may be created for these empty grid cells, whereas for a hashing function a number of empty data holes may be generated for these empty cells in the data file.

The decision as to which approach to use is an intuitive judgment depending on a consideration of the index file storage and the amount of storage to be reserved for empty grid cells in the partition. If the number of empty grid cells is very small then the hashing approach will be a better choice because it offers faster speed and the small number of empty data holes can offset the amount of storage required for the index file itself. Note however that, following selection of a primary strategy, further tuning processes are required to achieve near-optimal performance. This is

because the number of empty grid cells varies from application to application and changes during the course of database operations.

Research done in this area [LA88] suggests that for physical database organisation a specific implementation technique is usually suitable for a set of defined applications. If an application requires fast access time then it can employ a hashing algorithm to good effect. If, however, the response time is not critical and the volatility of data is high, a simple sequential file will be satisfactory. No single implementation will be the most-efficient for all applications because different applications may have different features, demanding individual consideration for their efficient use. It is proposed here that the expert system approach can be effectively employed in selecting a particular strategy of grid partitioning for a given application and further, that this approach can enhance the overall efficiency of the implementation.

This thesis considers a number of grid file algorithms and makes expert assessment of each according to its strengths and weaknesses based on an initial consideration of the various applications. These initial considerations fall into three categories: the data characteristics; the users' requirements; hardware and software constraints.

Chapter 2. Expert System Approach

Physical DB design is an extremely difficult task. Finding a good solution is difficult as the criteria for an optimal organisation cannot be exactly quantified. Database optimisers have been developed to reduce the computation and communication cost in a distributed environment. A variety of approaches have been mentioned in [SH91]. For example, the horizontal partitioning has been used in the context of distributed databases to increase the throughput and to save response time, the algebraic manipulations of query expressions transform a given query into an equivalent one that can be processed more efficiently; and the reorder of query conditions reduces the size of satisfactory relations for less complexity, etc.

An example of database optimiser is System R. System R performs optimisation of various operations over a database. In the optimisation process, System R makes a decision based on a cost estimation of different options. This estimation yields different computation and disk access costs. System R considers physical organisation of a relation which can affect the overall cost of a given query. The physical organisation of a relation provides information for the optimiser to take advantage of the indices. System R also allows relations to be stored according to its usage, providing storage flexibility so that logically related tuples can be accessed with low cost. System R* [MA86] is an extension of System R to the distributed environment. In addition to the strategies used in System R which optimises select, project, join and union, System R* has extra ability to handle replicated copies of a given relation in order to reduce transmission cost over the network. To perform the optimisation System R* [MA86] [M086] follows the following steps:

(1) generating all evaluation sequence;
(2) computing the best evaluation strategy;
(3) evaluating the cost of each option and
(4) selecting a strategy with the least cost.

The cost function used in System R* includes CPU, I/Os, and cost of transmission. In R* the transaction management uses two-phase locking protocol. Deadlock is allowed to occur and is resolved by deadlock detection and victim transaction abort.

Another example is Starburst [LI87] [CH90]. Starburst is an extension to existing database management systems such as INGRES, R, R* etc. The objective of Starburst is to facilitate the implementation of data management for relational databases. It provides alternative ways of storing relations (storage methods) and access paths, integrity constraints or triggers (attachments) to relations. It also provides support for diverse applications, i.e. supporting user-defined abstract data types and functions for fields of database records.

From the brief description of the DBMS optimisers, it can be seen that the physical organisation of a relation can greatly affect the cost of operations. The reason being that the number of I/Os required for a database operation depends on how data is stored to secondary storage device. A number of algorithms exist for implementing the grid partitioning approach. Each algorithm has its own strengths and weaknesses and for each algorithm one makes different expert opinions - heuristics - in selecting an algorithm for a specific application. It is also very difficult to define and classify an application with a clear pattern of its data and its query because the knowledge about it is fuzzy (it cannot be clearly quantified), dynamic (size and features change) and uncertain (changes cannot be predicted). The design of physical databases involves the determination of the data partitions, physical storage of data structures and access paths. These are based on a number of factors such as the block size, the types of query and the data distributions. Thus in practice, the selection of implementation algorithms is more likely done by heuristics whereby one matches the problem-dependent features of an application to the strengths of various grid implementation algorithms for a better solution. At the computer department of Erlangen-Nuernberg University, West Germany, an expert system has been built [AL87b] to support the configuration of VSAM (Virtual storage access method) file design. The objective of this system is to optimise VSAM file design by the implementation of a computer-aided file design system. In [AL87b], the author has recognised that VSAM file design is an application-orientated task. The expert system approach is used for the VSAM file organisation because the file designer needs to know a multitude of technical details in order to choose the right parameters, eg. bucket size, number of buffers, control area size, control interval size, etc. Tuning VSAM file design requires tradeoffs between various parameters. Further, how one chooses the values for these parameters depends upon the

features of an application.

A m-d physical DB design can be more complicated than a VSAM file design. As with VSAM file design, a m-d physical database design is an application-oriented task. To achieve optimal performance knowledge is required about the applications such as data distribution, data volatility, data set size, dimensionality of a data space, types of query and dominant query attributes. In addition, information is required about the system constraints such as the bucket size, available memory space and access modes. Moreover, heuristics are needed so that a valid choice can be made between the various implementation algorithms and data space partitioning strategies.

This thesis investigates factors relating to physical organisation of m-d data for the improvement of database performance. It assumes that:

(1) The database considered in this thesis are not distributed databases. The factors which influence the performance of distributed databases include relation partitioning and tuple distribution to different nodes over the network in order to reduce the cost of database operations. We assume that the transmission cost is not relevant to the physical organisation of data but relevant to distributing relations and scheduling the sequence of database operations.

(2) Performance is evaluated without the consideration of locking factors. We assume that the major influential factor of locking is the number of users accessing the same parts of a database concurrently. This mainly relates to the locking strategies used, and functionally partitioning and distributing relations in a database.

(3) Features of different data sets are derived from an equal-sized grid cell partition, i.e. the equal-sized grid cell partition is used to measure the features of all data sets for standardisation.

(4) Selecting one element from a group of tuples [BO90] which satisfy a query condition can be done by functional manipulations. For example, the author [FR86] has mentioned in his book (page 515) that basic sequence manipulating functions can be introduced to tackle the issue. The function $\alpha* \rightarrow \alpha$ returns

the first element of a sequence. we assume that the implication in the implementation of databases is the need of a control rule. This rule states that once there exists a tuple that matches the query conditions the result will be returned and the query operation terminated.

In the following chapter, the framework of a system model which uses heuristics to tune grid file design is discussed . We also show briefly how the system model works.

Chapter 3. The System Model

An expert system approach has been used for VSAM (Virtual Storage Access Method) file organisation [AL87b]. This section introduces the scenario of how an expert system approach can be applied to select physical organisations for multi-dimensional (m-d) data. A brief system framework is also described.

3.0. Introduction

The purpose of the system is to select an efficient implementation algorithm for a particular application and to provide monitoring and tuning facilities for a m-d physical database design. The requirements of the system are:

(1) to help users choose a near-optimal implementation of a physical database design;
(2) to monitor the performance of a database in order to satisfy users' requirements;
(3) to identify why a particular algorithm is chosen for a given application;
(4) to add new knowledge to the system, enabling the amendment and extension of the knowledge base and inference rules.

3.1. The Framework of the System Model

The characteristics of database applications are important factors which will influence the performance of databases. Researchers have developed various techniques for physical database organisation, catering for different applications [BU83] [EN88]. In practice, it is usually a mystery to the user why the performance deteriorates, eg. why the system gives a slower response at a certain stage. The reason is that the performance varies along with the changing applications (the pattern of the data and the frequency of queries change during the course of accessing and updating the database). The system model based on an expert approach is, therefore, designed to apply heuristics to match the characteristics of a given application to the strengths of an algorithm. It analyses the reasons behind any performance deterioration and works out what measures may be taken to improve the performance. In addition, it also takes the necessary actions to meet the users' requirements if possible.

The system model is made up of a user-system interface, database profiles, a knowledge base and a rule base. The knowledge base consists of an algorithm base, an application abstract profile base, a performance evaluation unit and a similarity comparison model. Its framework is illustrated in Figure 3.1.

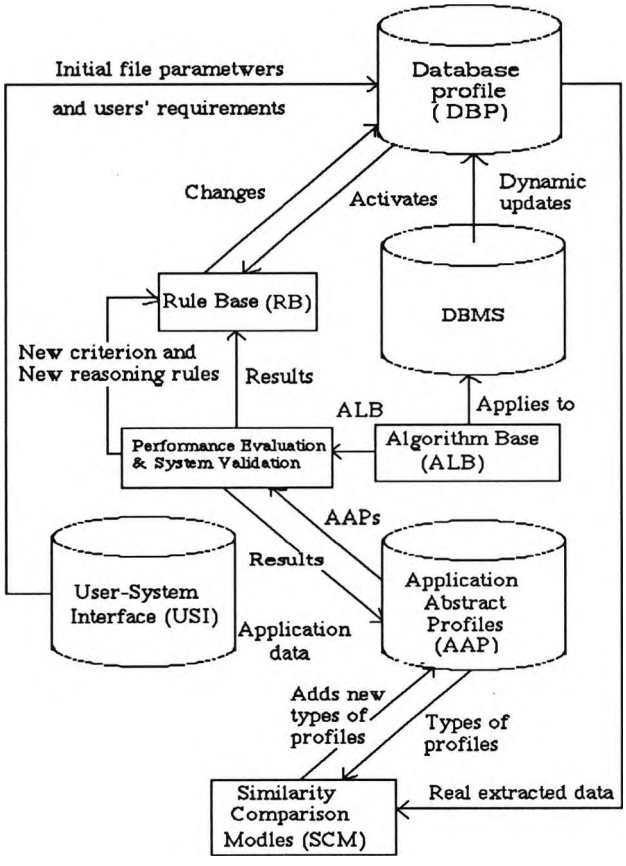


Figure 3.1 The System Framework.

A brief description of the system components

1. The User-System Interface (USI) captures the information from users about an initial knowledge of the database and system environment to construct an initial model of an application.

2. The Database Profile (DBP) stores information which describes the characteristics of an application.

3. The Algorithm Base (ALB) stores various alternative implementation schemes, splitting strategies, and merging strategies in terms of partitions and access paths.

4. The Similarity Comparison Model (SCM) carries out pattern matching in terms of data requirements and queries, in order to measure how closely a given application matches an existing application abstract profile.

5. The Application Abstract Profiles (AAP) store required information about data space partitions, data distributions, chosen implementation algorithms and query patterns. It assists in classifying applications. For instance, when the performance of the database falls to an unsatisfactory level in its performance the algorithm chosen for the AAP can be used to suggest a better implementation, provided a similar degree is identified to be satisfactory between the database profile and an application abstract profile (AAP).

6. The Performance Evaluation and System Validation (PESV) calculates speed and storage utilisation for a particular AAP. The system validation updates the rule base to achieve more accurate reasoning.

7. The Rule Base (RB) stores rules about algorithm selections and alterations to rules.

3.2. The Definition of the System Model

The system model is defined as:

S = {USI,DBP, K, RB}.

Where:

S - the System.

USI - User-System Interface.

DBP - the DataBase Profile stores relevant information about the physical database to be tuned. It consists of database features, including a set of the estimated initial database features obtained from the USI and a set of dynamic features acquired during the operations of the database. It also contains a set of parameters for each data set in the database:

DBP = {DBPi | for i = 1, 2, ..., x }

where x is the number of data sets which make up the database concerned.

The element DBPi (for i = 1, 2, ..., x) of DBP defines the characteristics of a data set i. These are the relevant properties of the database, including the initial and current database features - data patterns, query frequencies, system constraints, the chosen algorithm, and the corresponding performance.

K - the Knowledge required for the tuning process. This consists of the following:

K = (ALB, AAP, PESV, SCMs).

ALB - Algorithm Base.

AAP - Application Abstract Profile.

PESV - Performance Evaluation and System Validation.

SCM - Similarity Comparison Model.

RB - a Rule Base guides the system in deriving solutions or recommendations for operations. It is constructed on the basis of expertise, experience, heuristics, judgments, and individual decision criteria.

All the above information is stored as either factual or procedural knowledge: the factual knowledge gives the system evidence as a basis to guide selection and

matching processes; the procedural knowledge offers the system inference paths as an approach to deriving a solution.

Having defined the system, we now describe briefly how the system should work. As shown in Figure 3.1., when a new database is to be created, its initial data characteristics, such as the domain of the data space D, the data set size n, the dimensionality of the data space m, the users' requirements, hardware and software characteristics are obtained from the user-system interface via a dialogue between the system and users. Answers from users are validated and stored as initial characteristics of the application database system, producing an initial database profile. The database profile is used as the basis for selecting an implementation algorithm for the application concerned. The data characteristics are stored for a comparison of similarity between an application and an AAP. The users' requirements are employed as criteria by the system to trigger actions concerned with the monitoring of the database, and the hardware configuration indicates the constraints to the system. After an initial implementation algorithm has been selected the database will be physically stored on a secondary storage device according to this chosen algorithm. If the actual performance of a database does not meet the users' requirements at a certain stage it will trigger the system to either seek an alternative scheme to improve the situation, or indicate the relevant status of the current system. When the constructed* application profile matches an application type in the AAP base with required **similarity** (defined in the rule base) then using the similarity comparison model, the rule base will be searched to find a system-suggested implementation algorithm. If, on the other hand, the application profile cannot find an AAP that matches it with required similarity, then the application will be analysed and a new type of application may be added to the AAP base. Different stored implementation algorithms in the algorithm base will be applied to this new application type to carry out the performance evaluation through which the best algorithm for this new application type is chosen. In addition, the system should be able to update rules by itself. This update applies experimental

---

* To compare similarity between the two, information about an application needs to be constructed in accordance with the AAP

method and a "learn by expected result" approach. This is done by the Performance Evaluation and System Validation (PESV - see section 4.9 for details). The chosen implementation algorithm is then stored as knowledge in this new AAP as a system suggested solution. It is found that the data concerning the database may change during the life time of the database and so too may the characteristics of an application. These changes will be collected from the database optimiser (located in the DBMS), or the statistics of the database, or collected by the system itself. They are used to update the application database profile. When an unsatisfactory level of performance is detected, the system will extract the relevant information of the application profile, based on the new database profile, to compare the profile with the AAP. If an alternative solution is found in the system AAP then the system will adjust the database accordingly and take any necessary actions; otherwise, the system will display the hypotheses for the initial solution and the explanation of how the current database deviates from these hypotheses. We assume that the changing characteristics of the data as well as the query patterns are the source of any performance deterioration. By studying the dynamic behaviour of a database the causes of performance change may also be examined.

3.2.1. User-system Interface

The user-system interface is a means of capturing knowledge, relating to initial information concerning an application, from users. When a new data set is to be created the system will ask questions about the following aspects: the characteristics of a data set; users' requirements regarding the performance of the data set; hardware and software characteristics affecting the physical organisation of the data set. Some of the knowledge may be obtained from the database definition, such as data item length, access mode, cardinality, etc. In this case we can view the database definition as a user in terms of capturing initial knowledge about an application. An outline of the user-system interface is shown in Figure 3.2. As an implementation consideration, windowing technology can be applied to assist users to enter application features. A brief windowing USI is shown in Figure 3.3., and an initial algorithm selection is illustrated in Figure 3.4. respectively.

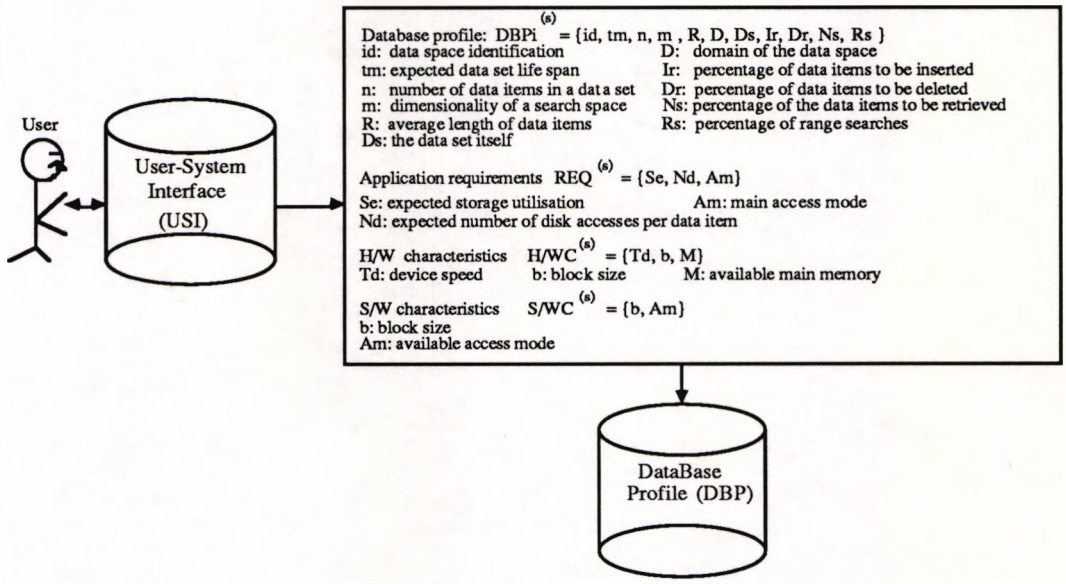


Figure 3.2. An outline of the user-system interface.

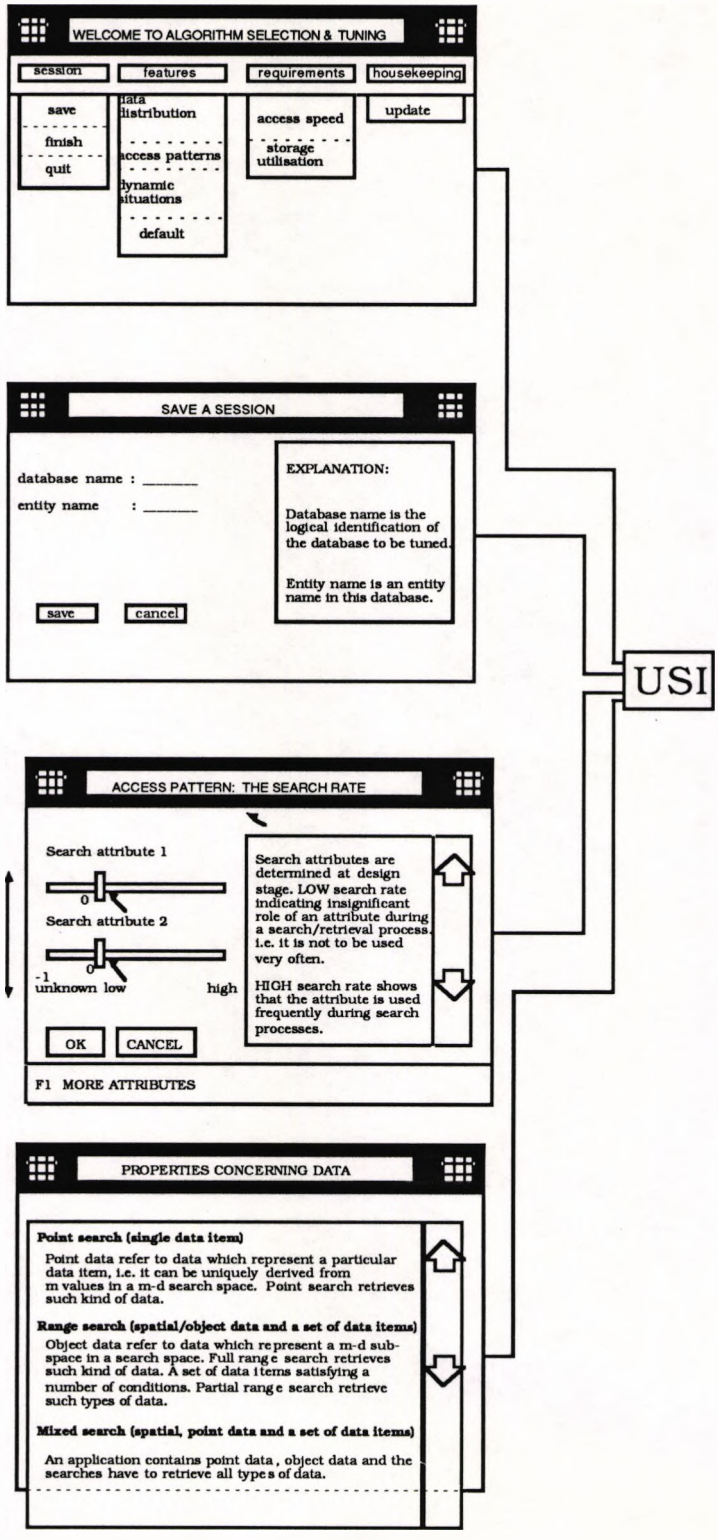


Figure 3.3 USI Implementation Consideration.

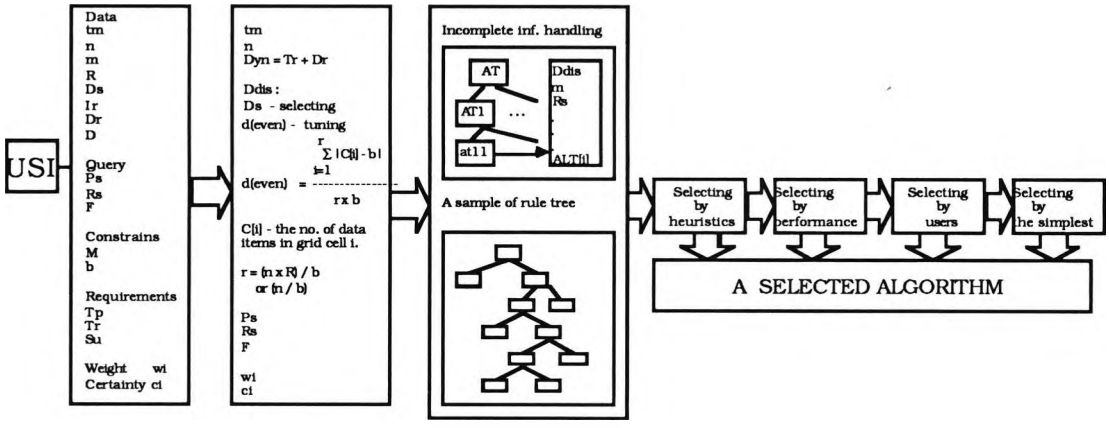


Figure 3.4 A Framework of Initial Algorithm Selection.

A user-system interface (USI) is defined as

$USI = \{ AC, DBP^{(s)}, REQ^{(s)}, H/WC^{(s)}, S/WC^{(s)} \}$

Here:

AC - Application Classification gives categories of applications of the AAP in the system, each application class including an application domain and its functionalities (tasks), for users to choose. The purpose of AC is to provide the system with the facility to process incomplete information. When the class of the concerned application matches the existing application class ACi the missing information will be extracted from the ACi in the AAP base. If there is no matched class in the system, a new one can be added by the users with its definition. The AC has two levels: a menu level for a user to select the corresponding application class; a definition level for users' reference. Each definition gives the meaning of an application class.

$AC = \{ ACi \mid \text{for } i = 1, 2, ..., x \}$

$DBP^{(s)}$ - Static Database Profile consisting of characteristics gained from the user-system interface.

$DBP^{(s)} = \{ DBP1^{(s)}, DBP2^{(s)}, ..., DBPx^{(s)} \}$

$= \{ DBPi^{(s)} \text{ for } i = 1, 2, ..., x \}$

$DBPi^{(s)}$ - initial assumptions about a data set that form the static features of the database profile.

$REQ^{(s)}$ - Users' Requirements.

$H/WC^{(s)}$ - Hardware Characteristics.

$S/WC^{(s)}$ - Software Characteristics: i.e. OS environment.

The detailed parameters for the above concepts are:

$DBPi^{(s)}$= {id, tm, n, m, R, D, Ds, Ir, Dr, Ns, Rs}

id - data space identification. In a relational database, a number of data sets will be created depending on the number of tables required for a database system. Each table represents a data space in the database system and each data space needs to be identified by a system number as an identifier in the USI.

tm - expected data set life span (how tm is measured needs experience). Taking a relational database as an example:

short - temporary and rarely used tables.

long - frequently used tables.

n - number of data items in a data set.

m - number of major attributes concerned with conducting searches (dimensionality of a search space).

R - average length of data items in a data set.

Ds - the data set itself:

Ds = { d1, d2, ..., dn}

di = (ai1, ai2, ..., aim) for i = 1, 2, ..., n.

di is data item i which consists of m attributes in terms of search space.

D - the range of each attribute concerned (i.e. the domain of the data space):

D = D1 X D2 X ... X Dm

Di = Dimax - Dimin for i = 1, ..., m.

where Dimax is the maximum value in the dimension i and

Dimin is the minimum value in the dimension i.

In geometric interpretations D forms a finite or bounded super-rectangle.

Ir - percentage of data items to be inserted (i.e. file growth rate)
_____ /n (%)

Dr - percentage of data items to be deleted (i.e. file shrink rate)
_____ /n (%)

Ns - percentage of data items to be searched (i.e. access rate)
_____ /run

Rs - percentage of range searches to be conducted (i.e. range search rate)
_____ /n

$REQ^{(s)}$ = {Se, Nsec, Am}

Se - expected storage utilisation (i.e. packing density).

Nsec - expected number of secondary storage accesses/per data item.

Am - main access mode (from database definition):
random
sequential
random access dominant say, random access > 50% of total queries.

$H/WC^{(s)}$ = {Td, b, M}

Td - device speed:
seek time - st,
transfer rate - tr.

b - block size:
_____ bytes or b data items

M - available memory for the data set (for a decision on whether an index can be accommodated in the memory or not)

$S/WC^{(s)}$ = {b, Am}

b - block size
_____ bytes OR b data items

The USI also provides facilities for user to ask system questions about reasoning regarding an application at different levels of details.

3.2.2. Dynamic changes to the database profile

The dynamic changes to the database profile are summarised in Figure 3.5.

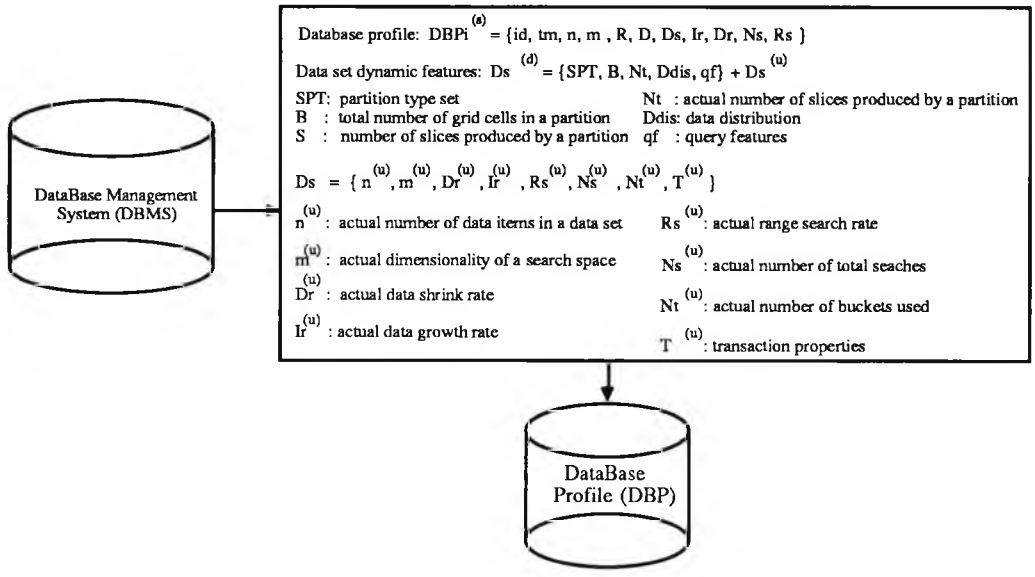


Figure 3.5. The dynamic changes to the database profile.

We present these changes as follows:

$DBP^{(d)}$ = { $DBP1^{(d)}$, $DBP2^{(d)}$ ..., $DBPx^{(d)}$ }

= { $DBPi^{(d)}$ for i = 1, 2, ..., x }

where:

$DBPi^{(d)} = DBPi^{(s)} + Ds^{(d)}$

for i = 1, ..., m

$DBPi^{(s)}$ = {id, tm, n, m, R, D, Ds, Ir, Dr, Ns, Rs}, they are defined in section 3.2.1

$Ds^{(d)}$ $= \{ SPT, r, S, Nt, Ddis, qf \} + Ds^{(u)}$

SPT - selected partition type.

$SPT \in PT = \{PT1, PT2, ... \}$

PTi - partition type i. Several partition types can be applied to a data set, but only one is in use at any time frame.

r - total number of grid cells in a partition.

S - number of slices produced by a partition for each dimension:

$S = \{ s1, s2, ... , sm \}$.

A slice is the area enclosed by a partitioning boundary.

Nt - total number of buckets actually used.

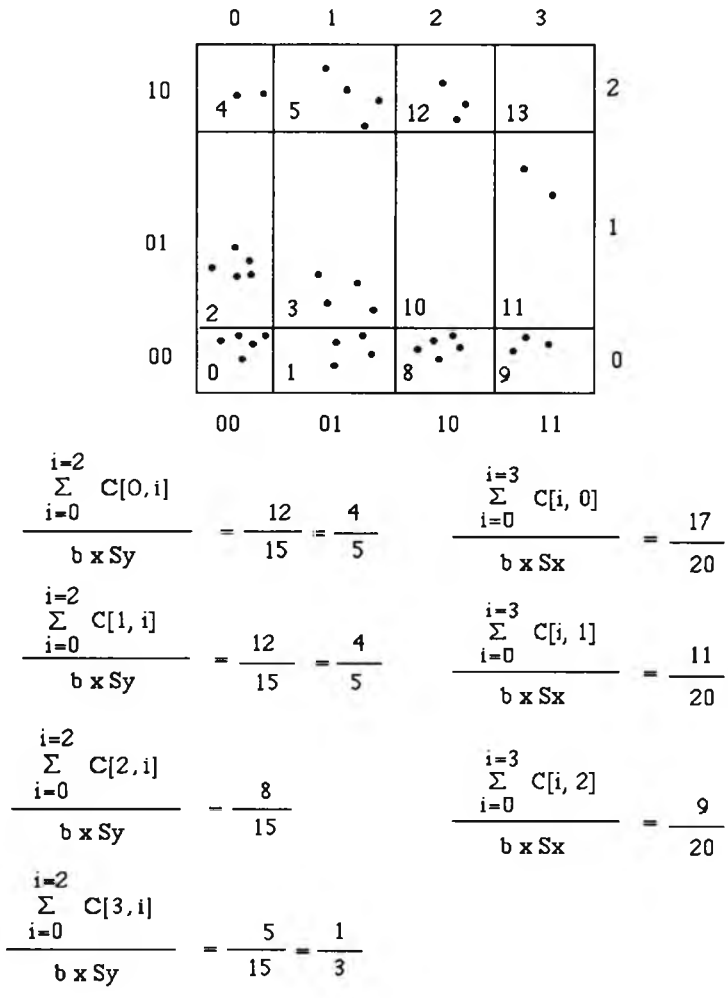
Ddis - data distribution.

Ddis is measured by $\{ C[i] \mid \text{for } i = 1, 2, ..., r \}$

here C[i] is the number of data items in the grid cell i.

C[i] - number of data items of a grid cell i in a partition. i is a z-code (it will be elaborated later) corresponding to the grid cell concerned.

Based on the value in C[i], the local data density can be calculated to guide a splitting process. The calculation is shown in Figure 3.6.



$$\frac{\sum_{i=0}^{i=2} C[0,i]}{b \times Sy} = \frac{12}{15} = \frac{4}{5} \qquad \frac{\sum_{i=0}^{i=3} C[i,0]}{b \times Sx} = \frac{17}{20}$$

$$\frac{\sum_{i=0}^{i=2} C[1,i]}{b \times Sy} = \frac{12}{15} = \frac{4}{5} \qquad \frac{\sum_{i=0}^{i=3} C[i,1]}{b \times Sx} = \frac{11}{20}$$

$$\frac{\sum_{i=0}^{i=2} C[2,i]}{b \times Sy} = \frac{8}{15} \qquad \frac{\sum_{i=0}^{i=3} C[i,2]}{b \times Sx} = \frac{9}{20}$$

$$\frac{\sum_{i=0}^{i=2} C[3,i]}{b \times Sy} = \frac{5}{15} = \frac{1}{3}$$

Figure 3.6. Calculate the local data density for each slice.

In this diagram there are four slices on the x dimension and three slices on the y dimension respectively. Consequently, x varies from 0 to 3 (4 slices) and y ranges from 0 to 2 (3 slices) in the calculations shown in Figure 3.6.

qf - query features.
qf = (Qt, W = (Wi | for i = 1, ..., m))
for a 2-d data space m = 2.
Where:
Qt - Query type;
W - Properties of attribute set by weighting;

The weighting is derived from the frequencies of each attribute, i.e. if attribute A is a dominant one for a set of queries then more splits may take place to increase the proximity along dimension A. We will discuss the frequency controlled partitioning in section 4.1.

$Ds^{(u)} = \{n^{(u)}, m^{(u)}, D^{(u)}, Ir^{(u)}, Dr^{(u)}, Rs^{(u)}, Ns^{(u)}, Nt^{(u)}, T^{(u)}\}$

where

$n^{(u)}$ - the number of data items in the data space considered up-to-date.

$m^{(u)}$ - the number of dimensions of the search space up-to-date.

$Ir^{(u)}$ - actual data growth rate.

$Dr^{(u)}$ - actual data shrink rate.

$Rs^{(u)}$ - actual range search rate.

$Ns^{(u)}$ - actual number of total searches.

$Nt^{(u)}$ - actual number of buckets used.

$T^{(u)}$ - transaction properties.

$Ds^{(u)}$ is a set of parameters which reflects the current status of the data set. Whether the system should update when there is a change or store the changes separately is a system decision.

3.2.3. The Database Profile

The profile information about a database gives a complete picture about an application. This profile information, made up profiles of data, queries, constraints

and performance, comes from three channels - user-system interface, performance evaluation, and database optimiser or statistics captured by the system itself. The framework of the profile is shown in] 3.7. These parameters, which describe the profile, are given below.

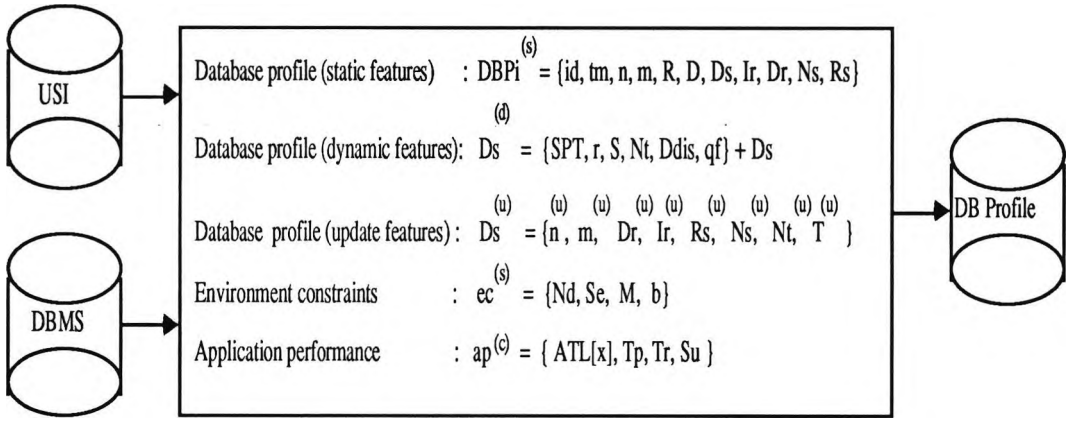


Figure 3.7. The framework of the profile.

In the diagram, we have:

DBP = { DBP1, DBP2, ..., DBPx }

DBPi = { $DBPi^{(d)}$, $ec^{(s)}$, $ap^{(c)}$ } for i = 1, ..., x

$DBPi^{(d)}$ = $Ds^{(s)}$ + $Ds^{(d)}$

$DBPi^{(s)}$ = {id, tm, n, m, R, D, Ds, Ir, Dr, Rs, Ns}

$Ds^{(d)}$ = { SPT, B, S, Nt, Ddis, qf } + $Ds^{(u)}$

$Ds^{(u)}$ = {$n^{(u)}$, $m^{(u)}$, $D^{(u)}$, $Ir^{(u)}$, $Dr^{(u)}$, $Rr^{(u)}$, $Rs^{(u)}$, $Nt^{(u)}$,$T^{(u)}$}

$ec^{(s)}$ - environment constraints.

$ec^{(s)}$ = { Nd, Se, M, b }

$ap^{(c)}$ - application performance.

$ap^{(c)}$ = { ALT[x], Tp, Tr, Su }

ALT[x]	-	the chosen algorithm.
t1	-	time estimated for point search.
t2	-	time estimated for range search.
Su	-	storage utilisation.
$T^{(u)}$	-	transaction properties.

Here (d) indicates the dynamic characteristics of the database profile, "(c)" indicates the calculated features of the database profile, "(u)" indicates the update features of the database profile, and "(s)" indicates the static features of the database profile, which has been described in section 3.2.1. concerning the user-system interface.

Chapter 4 The Knowledge Base

This section is the main part of this thesis. In this part the components of knowledge base are described. Firstly, various m-d access algorithms are analysed. Essential design features (speed, storage utilisation, flexibility of dealing with dynamic situations, ability of handling varied data distributions) of each are discussed. Desirable features for m-d hashing algorithms are matched with specific application characteristics, providing a rule base to drive the knowledge system in assisting database designers to choose a suitable algorithm for his/her application. Secondly, heuristically matching application abstract profiles to new applications are also discussed. Finally, performance issues are analysed.

4.0 Framework of Knowledge System

In order to tune physical database design the system is constructed to include several categories of knowledge which we describe below. The framework of the knowledge base is shown in Figure 4.1.

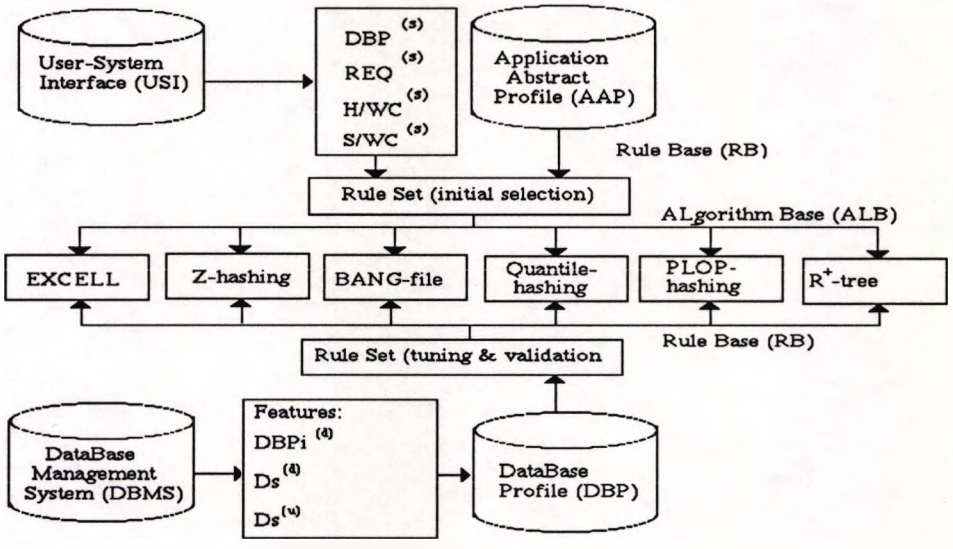


Figure 4.1. The knowledge base framework.

4.1. The Algorithm Base

The algorithm base is a collection of the implementation schemes used to evaluate the performance for different AAPs. Hence, there are constructed in a way which facilitates the performance evaluation.

There are a variety of implementation algorithms for grid partitions. These algorithms are: k-d-Tree [RO81], hB-Tree [LO87], R-Tree [GU84], R^+-Tree [SE87], BANG file [FR86] [FR87] [FR88], EXCELL scheme [TA82], EXHASH scheme [TA82], quantile-hashing [KS88b] [KS87], PLOP-hashing[KS88a], Gray code z-ordering [FA86b], binary code z-hashing [HU88a], etc. Each algorithm performs the same task: organising data onto secondary storage in such a way that upon a given query the required data item(s) can be extracted from the stored data set. These algorithms mainly differ in the way they carry out data space partitioning (splitting and merging), where data structures are applied to represent (or to order) the relationship between individual components of the partitioning, also they differ in the way they store access paths. For example, the z-hashing and EXCELL algorithms share the same kind of partitioning strategy. However, they use different data structures for storing the access paths for the partitioning. The BANG file and hB-Tree algorithms employ the same type of partitioning strategy. Both techniques aim to increase the efficiency of dealing with non-uniform data distributions, in the mean time, they use different approaches to represent the partition and it's access paths. Each of these algorithms differs in complexity as well as performance behaviour according to different characteristics of applications. The same partition with different access paths implementation, or the same paths implementation applying to different partitions, makes for the different algorithms. Essentially the partition is classified as a dimension-simultaneous (the Bang file and the hB-tree) partition and a dimension-alternate (the EXCELL and z-hashing) partition, each involving variants. The access paths implementation is mainly divided into two categories: hashing and indexing. We will choose combinations of five kinds of partitions and four types of access path strategies to be representatives from the above mentioned algorithms to construct our algorithm base. In addition, we will describe the heuristics used to select an algorithm for a particular application.

For simplicity, we will concentrate on the 2-d search space for all implementation algorithms so that they can be modelled by two axes, x and y. For higher

dimensions, the algorithms can be generalised for all aspects except the performance evaluations. The performance feature can be changed dramatically when the dimensionality increases, similar to a relation in a relational database using a multi-attributes index, as the number of attributes increases heavy storage and access overhead is likely to be the result .

Three aspects are considered here in order to describe an implementation algorithm: (1) the partition types, (2) the strategies for coping with insertions and deletions, and (3) the storage of data access paths.

(1) Partition types

PT1 Partitioning a data space into equal-sized grid cells

In this partition each split will double the number of the grid cells. Each grid cell is represented by a pair (i, j) where i is the i^{th} slice on the x axis and j is the j^{th} slice in the y dimension. These slices are created by the equal-sized grid cell partition. The partitioning process is shown in Figure 4.2.

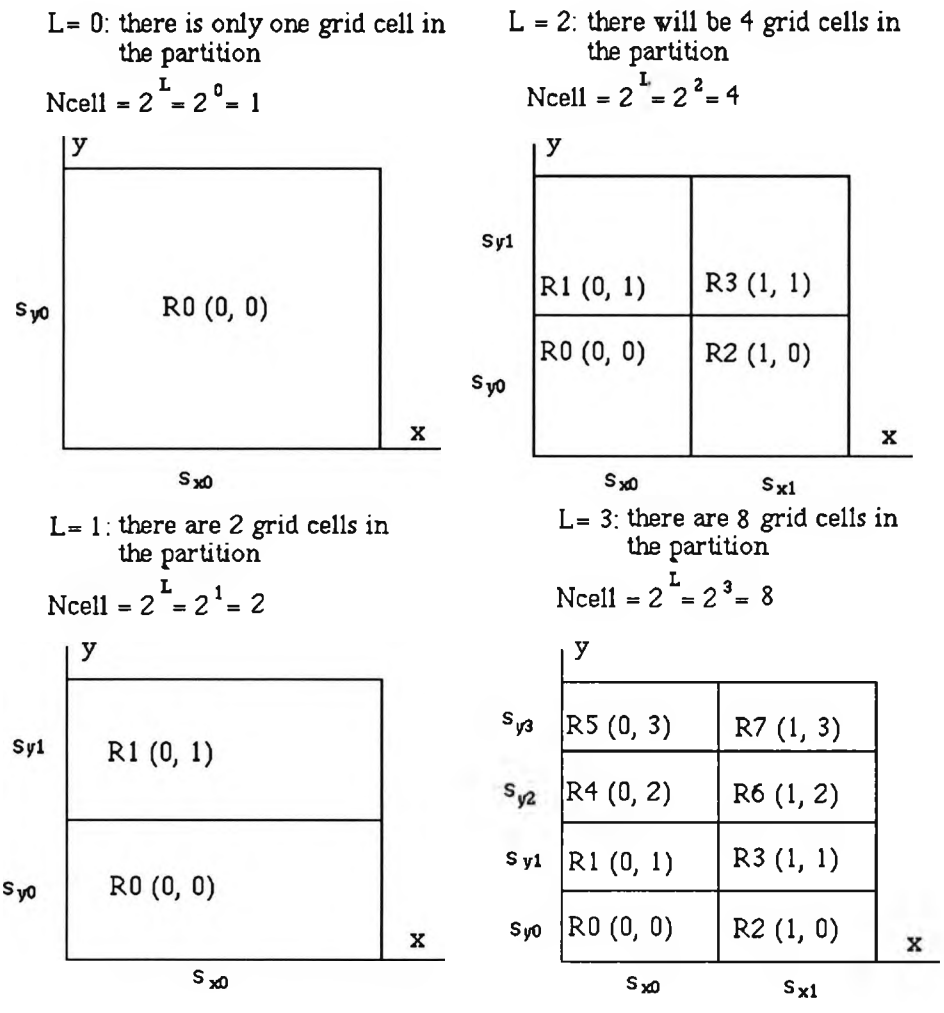


Figure 4.2. Equal sized grid cell partition.

In the diagram, each grid cell forms a rectangle numbered by z-code (Rz) such as R0 = (0, 0), R1 = (0, 1), R2 = (1, 0) and so on. The z-code will become apparent when it is used later.

Since every grid cell created by the partition has to be equal in size a split in the k^{th} dimension ($k \in 1, 2, ..., m$) will automatically produce another sk (numbered by 0, 1, ..., sk -1) slices in the k^{th} dimension. Thus the number of slices in the k^{th} dimension is doubled. If we use $Ncell^{(b)}$, $Ncell^{(a)}$ to represent the number of grid cells before and after a split and $sj^{(b)}$ and $sj^{(a)}$ (for j = 1, 2, ..., m) to represent the number of slices in the j^{th} dimension before and after a split respectively then, after a split in the j^{th} dimension we get:

$$Ncell^{(b)} = \prod_{j=1}^{j=m} sj^{(b)}$$

$$Ncell^{(a)} = \prod_{j=1}^{j=m} sj^{(a)}$$

$$= 2 \times \prod_{j=1}^{j=m} sj^{(b)} \qquad \text{(every cell split into two cells)}$$

$$= 2 \times Ncell^{(b)} = Ncell^{(b)} + Ncell^{(b)}$$

Except that $sk^{(a)} = 2 \times sk^{(b)}$ other items remain unchanged after a split in the j^{th} dimension and therefore, the number of grid cells added after a split is the second item in the above formula. A split doubles the number of grid cells.

PT2 Partially equal sized partitioning of a search space

In this partition at each split the number of grid cells increases at the rate of the number of slices on the orthogonal dimension(s) of the split dimension. If a split happens in the k^{th} dimension then the relation $sk^{(a)} = sk^{(b)} + 1$ holds. Thus we

get:

$$Ncell^{(a)} = \prod_{j=1}^{j=m} s_j^{(a)}$$

$$= \prod_{\substack{j=1 \\ j\neq k}}^{j=m} s_j^{(b)} \times (s_k + 1)$$

$$= \prod_{j=1}^{j=m} s_j^{(b)} + \prod_{\substack{j=1 \\ j\neq k}}^{j=m} s_j^{(b)}$$

$$= Ncell^{(b)} + \prod_{\substack{j=1 \\ j\neq k}}^{j=m} s_j^{(b)}$$

The number of grid cells added after a split in the dimension is the second item in the above formula. This item indicates the number of grid cells affected by a split and the number of grid cells added has reduced comparing with PT1 especially when m is large. The partition process is shown in Figure 4.3.

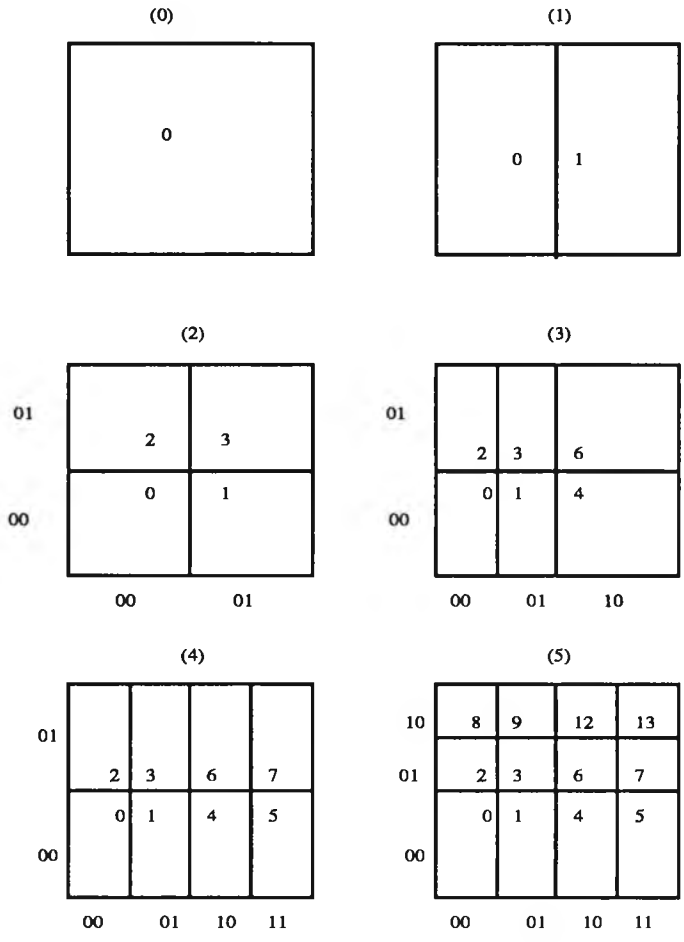


Figure 4.3. Partially equal sized grid cell partition.

PT3 Parameter guided partitioning of a data space

For each split in this partition, the local density or the frequency of the attributes is used to decide the partition position or split frequencies in each dimension.

There are two different groups of control rules for the same type of partition. The first group is applied without overflow handling. In this situation, when an overflow occurs there will not be enough room to accommodate all data items because the bucket size b is fixed. As a result, a split has to be triggered. The second group is used with overflow handling. In this circumstance, when overflow occurs those data items which cannot fit in the home grid cell may be stored in chained data buckets or stored according to some other overflow handling techniques. In such situations, an upper bound (threshold) is given to determine when a split is going to be triggered.

PT 3.1. Local density guided partitioning

The local density is denoted by Ld(i,j), which indicates the packing density in the j^{th} slice and in the i^{th} dimension. Two circumstances apply:

(a) without overflow handling

When overflow is not handled by the system then we are forced to split the grid cell if the required storage (indicated by the number of data items falling in a grid cell) exceeds the bucket size (i.e.block size b). The splitting process is fired by the rule:

if the added data item is in the grid cell identified by [a1, a2, ..., am] and

if $\max_{1 \le i \le m} (Ld(i,k) \text{ for } i = 1, 2, ..., m, k = a1, a2, ..., am) = Ld(i,ax)$

then divide on the $(ax)^{th}$ slice in the i^{th} dimension.

For instance, if m = 2 (say, with dimensions x, y) and the chosen dimension to be split is x then jk = sy; if m = 3 (eg. with dimensions x, y, z) and the chosen dimension is x then jk = sy or jk = sz. The geometrical meaning is shown in Figure 4.4. (a), where C[i1, i2, i3, ..., im] is the number of data items within the super-rectangle identified by [i1, i2, ..., im]. Using [i1, i2, ..., im] a z-code is generated as the subscript of each count. Hence the representation C[i1, i2, ..., im] is equivalent to C[z] where z = z-code for [i1, i2, ..., im]. The condition as to which slice to split next depends on the overflowed grid cell. In addition, it depends on the values of local density measured for these slices that intersect with this overflowed grid cell. Hence when the added data item causes an overflow in the grid cell

identified by [a1, a2, ..., am] there will be m alternative slices to choose from. Using the local density guided split strategy the one with the maximum value for its local density will be chosen. For example, in a 2-d data space if the inserted data item is within the range of grid cell [ax, ay] and, before splitting, there are sx and sy slices on the x and y axes respectively, then, to calculate the density in the x dimension, we need to choose y = 1, 2, ..., sy to form the local density value for the slice sx. In the x dimension:

$$Ld(ax) = \frac{\sum_{y=0}^{y=sy-1} C[ax, y]}{b \times sy}$$

Similarly, in the y dimension:

$$Ld(ay) = \frac{\sum_{x=0}^{x=sx-1} C[x, ay]}{b \times sx}$$

For this example, the choice of a slice to split depends on the maximum function, $\max_{1 \le i \le m} (Ld(ax), Ld(ay))$. If Ld(ax) > Ld(ay) then the ax slice should be split in the x dimension otherwise the slice ay should be split in the y dimension.

C[x, y] refers to the C[z-code], where z-code is the result of interleaving the binary form of x and y alternately. The partitioning process and its z-code numbering is shown in Figure 4.4.(a).

(b) with overflow handling

When overflow handling is considered, a value can be set to the local density level as a threshold to control the splitting or merging process. The splitting, therefore, can be triggered by the following rule:

If $\max_{1 \le i \le m} (Ld(i,j) \text{ for } j = 1, 2, ..., sj) = Ld(i,k)$, where $k \in (1, 2, ..., sk)$ and

$$\prod_{\substack{j=1 \\ i \neq j}}^{j=sj} sj \times b \leq Ld(i, k) \times (b \times si) \quad \text{(the left hand side - LHS)}$$

then a split will take place on the k^{th} slice in the i^{th} dimension (the right hand side - RHS). We can see from the above formula that there are

$$\sum_{k=1}^{k=m} sk$$

alternative slices to choose from for a split or a merge.

For i^{th} slice, data density can be calculated as:

$$Ld(i, k) \times (b \times si) = \sum_{j1=0}^{j1=sj1-1} \sum_{j2=0}^{j2=sj2-1} \cdots \sum_{jm=0}^{jm=sjm-1} C[j1, j2, ..., jk, ..., jm]$$

where jk is a constant, jk = i .

The item in the right hand of the equal sign is formed in a similar way to sum all elements in a multi-dimensional matrix for k^{th} column, where each element is assigned to the number of data items in a grid cell which is numbered by [j1, j2, ...,i, ..., jm].

Here the sjk (for k = 1,2, ..., m; $k \neq i$) is the number of slices in the orthogonal dimension(s) to the i^{th} dimension to be divided. For instance, if m = 2, then x, y are two dimensions and (sx, sy) are the number of slices in the x and y dimensions respectively. Thus in the x dimension:

$$Ld(1,y) = \frac{\sum_{y=0}^{y=sy-1} C[1, y]}{b \times sy}$$

where x always equals 1.

$$Ld(2,y) = \frac{\sum_{y=0}^{y=sy-1} C[2,y]}{b \times sy}$$

here x always equals 2.

.
.
.

$$Ld(sx,y) = \frac{\sum_{y=0}^{y=sy-1} C[sx,y]}{b \times sy}$$

where x is a constant sx.

and similarly, in the y dimension:

$$Ld(1,x) = \frac{\sum_{x=0}^{x=sx-1} C[x,1]}{b \times sx}$$

$$Ld(2,x) = \frac{\sum_{x=0}^{x=sx-1} C[x,2]}{b \times sx}$$

.
.
.

$$Ld(x,sy) = \frac{\sum_{x=0}^{x=sx-1} C[x,sy]}{b \times sx}$$

Figure 4.4. (b) shows the situation.

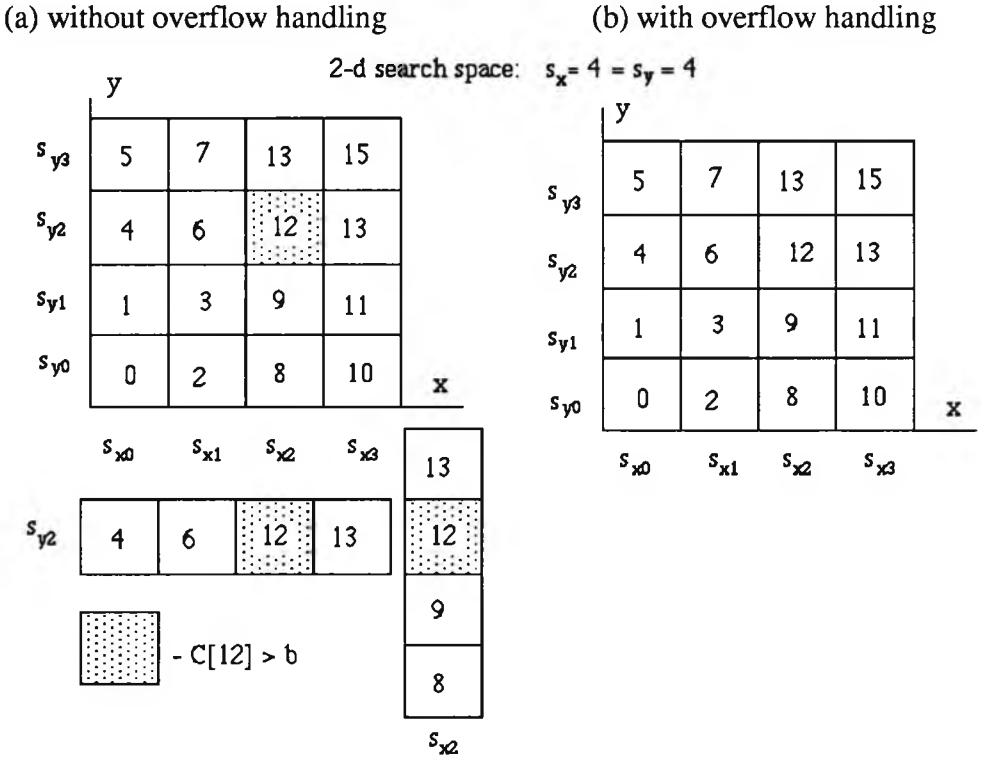


Figure 4.4. Local density controlled grid partition.

The split control for case (a) is: max (Ld(sx2), Ld(sy2)). Where

$$Ld(sij) = \frac{\sum_{i=0}^{i=si-1} C[i]}{b \times si}$$

i = x or y
j = 2

The split control for case (2) is: max (Ld(sxj), Ld(syj) | for j = 0, 1, 2, 3). Where

$$Ld(sxj)_{j=0,1,2,3} = \frac{\sum_{j=0}^{j=sj-1} C[i,j]}{b \times sj}$$

$$Ld(syi)_{i=0,1,2,3} = \frac{\sum_{i=0}^{i=si-1} C[i,j]}{b \times si}$$

In Figure 4.4, for case (a) there are only two slices to be chosen from (sx2 and sy2) because no overflow is allowed. Thus it will force the grid cell 12 to split as C[12] > b, here b is the bucket size. For case (b) there are $\sum_{j=x,y} sj = 4 + 4 = 8$ slices to be chosen from.

PT3.2: query frequency / attribute weight (Wi) guided partitioning

(a) with overflow handling

If ($\max_{1 \le i \le m}$ (Wi) = on j^{th} dimension for i = 1, 2, ..., m)

and $\left(sj \le \frac{Wj}{Wmin} \times \frac{\sum_{k=1, k \neq j}^{k=m} sk}{m-1}\right)$ then divide on the j^{th} dimension.

where Wmin = min (W1, W2, ..., Wm).

Note though, when W1 ≈ W2 ≈ ... ≈ Wm the frequency control is no longer effective because this condition indicates the indifference of Wi.

(b) without overflow handling

The frequency guided partition, which is similar to the local density partition, has m alternative dimensions to choose from when a split occurs. If the added data item is located in the grid cell identified by [a1, a2, ..., am] then the following rule can be applied to guide the splitting. If (max (Wi) = Wj in the j^{th} dimension for i = 1, 2,

$1 \le i \le m$

..., m) and

$$(s_j \leq \frac{W_j}{W_{min}} \times \frac{\sum_{k=1,k\neq j}^{k=m} s_k}{m-1})$$ then divide the $(a_i)^{th}$ slice in the j^{th} dimension.

We have seen that for both cases there are two conditions which must be satisfied when choosing a split strategy.

C1 = $(\max_{1\leq i\leq m}(W_i) = j^{th}$ dimension for i = a1, a2, ..., am)

$$C2 = (s_j \leq \frac{W_j}{W_{min}} \times \frac{\sum_{k=1,k\neq j}^{k=m} s_k}{m-1})$$

The first condition is used to determine the dimension based on query frequencies and the second condition decides whether the number of splits (i.e. number of slices) in the chosen dimension has reached its required frequency for the partition. The partitioning processes are shown in Figure 4.5. (a1, a2, b1, b2) respectively.

Query frequency controlled grid file partition illustrations

(a) with overflow handling

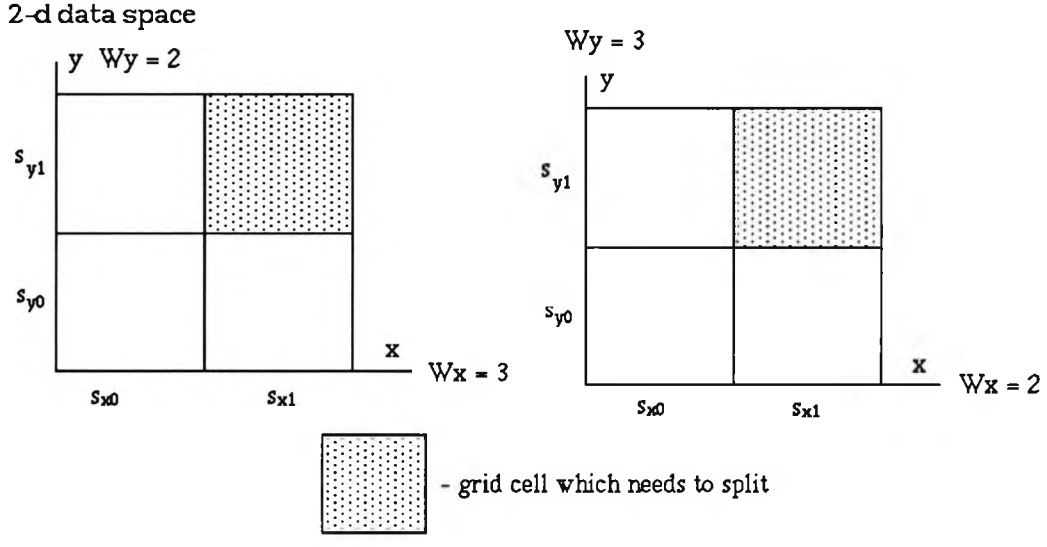


Figure 4.5. (a1) The dimension chosen by the frequency function has not reached its split frequency.

In this diagram we have:

(a) max (Wx, Wy) = Wx

min (Wx, Wy) = Wy

sx = 2 (before a split)

$$\frac{Wx}{Wy} \times \frac{sy}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sx \leq \frac{Wx}{Wy} \times \frac{sy}{m-1}$$

The split is therefore, as follows: if Ld(sx0) > Ld(sx1) then split sx0 else split sx1;

(b) max (Wx, Wy) = Wy
min (Wx, Wy) = Wx
sx = 2 (before a split)

$$\frac{Wy}{Wx} \times \frac{sx}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sy \leq \frac{Wy}{Wx} \times \frac{sx}{m-1}$$

The split is therefore, as follows: if Ld(sy0) > Ld(sy1) then split sy0 else split sy1.

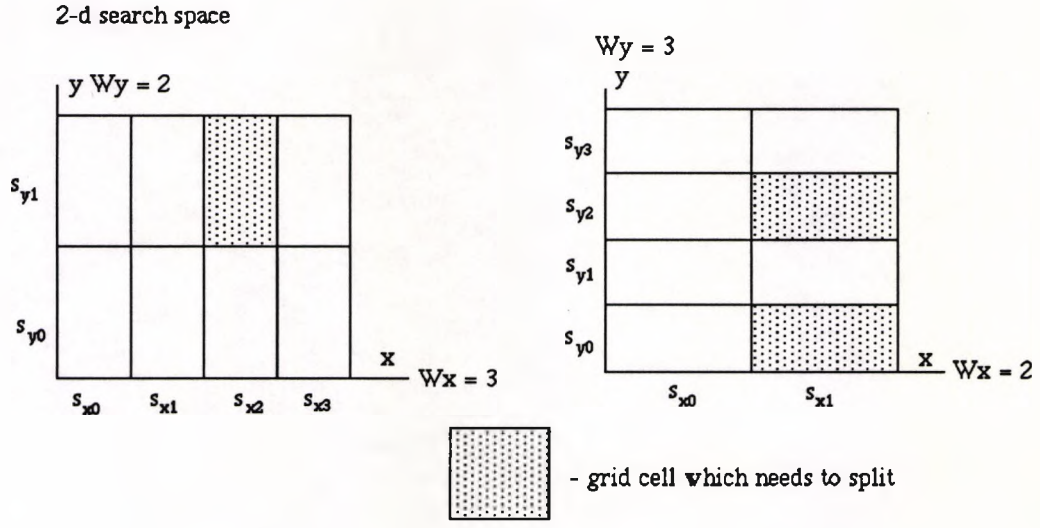


Figure 4.5. (a2) The dimension chosen by the frequency function has already reached its split frequency.

In this diagram we have:

(a) max (Wx, Wy) = Wx
min (Wx, Wy) = Wy
sx = 4 (before a split)

$$\frac{Wx}{Wy} \times \frac{sy}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sx > \frac{Wx}{Wy} \times \frac{sy}{m-1}$$

Even if Wx > Wy the next split should take place in y dimension, i.e. the split follows: if Ld(sy0) > Ld(sy1) then split sy0 else split sy1.

(b) max (Wx, Wy) = Wy
min (Wx, Wy) = Wx
sy = 4 (before a split)

$$\frac{Wy}{Wx} \times \frac{sx}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sy > \frac{Wy}{Wx} \times \frac{sx}{m-1}$$

Even if Wy > Wx the next split takes place in the x dimension, i.e.
if Ld(sx0) > Ld(sx1) then split sx0 else split sx1.

(b) without overflow handling

In the following diagram we have:

(a) max (Wx, Wy) = Wx
min (Wx, Wy) = Wy
sx = 2 (before a split),

$$\frac{Wx}{Wy} \times \frac{sy}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sx \leq \frac{Wx}{Wy} \times \frac{sy}{m-1}$$

The new split is in the x dimension.

(b) max (Wx, Wy) = Wy
min (Wx, Wy) = Wx
sy = 2 (before a split)

$$\frac{Wy}{Wx} \times \frac{sx}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sy \leq \frac{Wy}{Wx} \times \frac{sx}{m-1}$$

The new split is in the y dimension.

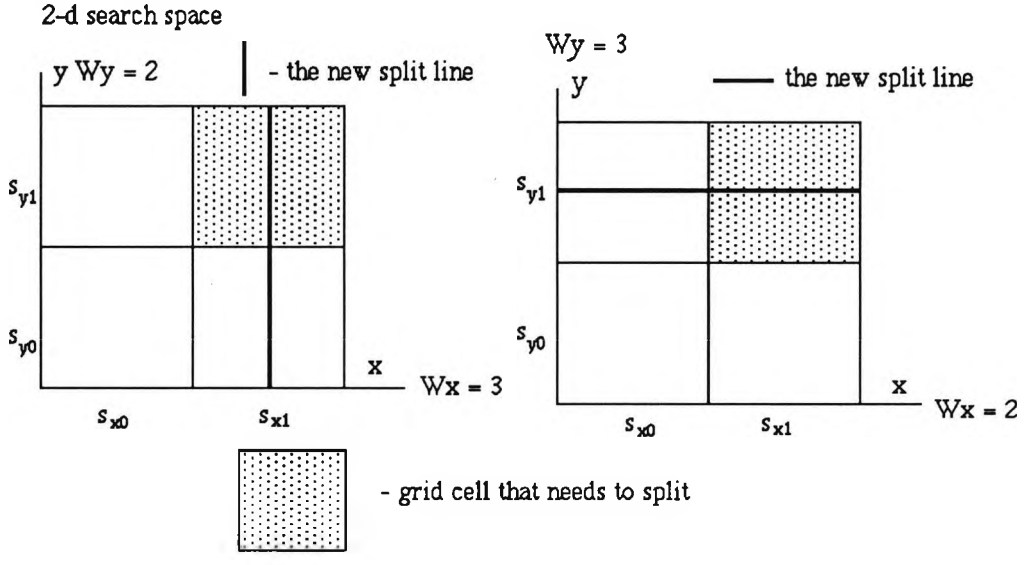


Figure 4.5. (b1) the dimension chosen by the frequency function has not reached its split frequency.

In the diagram below we have:

(a) max (Wx, Wy) = Wx
min (Wx, Wy) = Wy
sx = 4 (before a split),

$$\frac{Wx}{Wy} \times \frac{sy}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sx > \frac{Wx}{Wy} \times \frac{sy}{m-1}$$

Even if Wx > Wy the next split should take place in y dimension.

(b) max (Wx, Wy) = Wy,
min (Wx, Wy) = Wx,
sy = 4 (before a split),

$$\frac{Wy}{Wx} \times \frac{sx}{m-1} = \frac{3}{2} \times \frac{2}{1} = 3$$

thus

$$sy > \frac{Wy}{Wx} \times \frac{sx}{m-1}$$

Even if Wy > Wx the next split takes place in the x dimension.

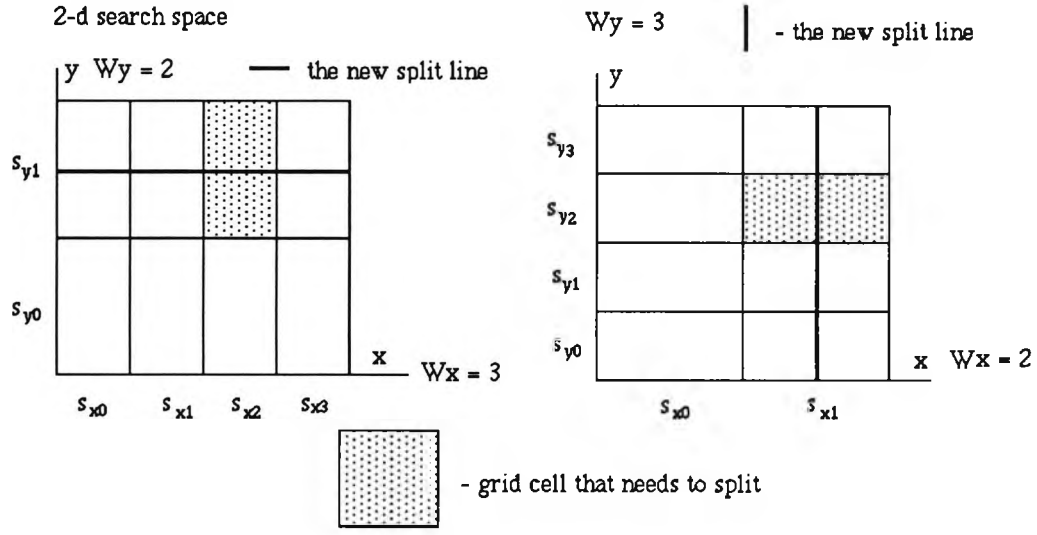


Figure 4.5. (b2) The dimension chosen by the frequency function has already reached its split frequency.

PT3.3 Query and density mix guided partitioning

(a) with overflow handling

If ($\max_{1 \le i \le m}$ (Wi) = Wd, in the d^{th} dimension for i = 1, 2, ..., m)

$$\text{and } \left(sd \le \frac{Wd}{Wmin} \times \frac{\sum_{\substack{k=1 \\ k \neq d}}^{k=m} sk}{m-1}\right)$$

and ($\max_{1 \le i \le m}$ (Ld(d,j) for j = 1, 2, ..., sj) = x)

then divide on slice x of the j^{th} dimension if possible otherwise determine the split on local density Ld.

(b) without overflow handling

If the inserted data item is in the grid cell of [a1, a2, a3, ..., am] then the following rule is applied:

If ($\max_{1 \le i \le m}$ (Wi) = Wd, in the d^{th} dimension for d = 1, 2, ..., m)

$$\left(sd \le \frac{Wd}{Wmin} \times \frac{\sum_{\substack{k=1 \\ k \neq d}}^{k=m} sk}{m-1}\right)$$

then divide on the $(ad)^{th}$ slice of the j^{th} dimension if possible, otherwise divide based on the value of the local density:

$\max_{1 \le i \le m}$ (Ld(i,j)) for i = 1, 2, ..., m; j = a1, a2, ... am.

It will be evident that for all types of partition categorised in PT3, the dividing points in each dimension have to be stored because they may not be equal in range and frequencies (i.e. the grid cells may have different sizes and the number of divisions in each dimension may also differ). Thus, to obtain better partitioning the

storage of extra information is needed. Consequently, to choose between equal sized grid cell partition and local density or query frequency controlled grid partition there is a storage tradeoff to be made. Two situations have been discussed for these partitions - with and without overflow handling. Whether or not to use overflow handling is determined by data distribution. An even and high growth rate data set may be organised by applying no overflow handling, because the sparsely populated grid cells generated by a new split can be quickly filled; whereas an uneven and low growth rate data set may need to employ overflow handling. Using an expert system to tune the physical database makes it possible for different strategies to be used for the same data set at different times.

PT4 Independent data distribution controlled partition

To describe this partition we assume that the data distribution is independent for each attribute. Let us consider a 2-d data space x, y. A data set can be represented by Ds = {d1, d2, ..., dn}, di = (xi, yi) for i = 1, 2, ..., n. If f(xi) and f(yi) are the distribution functions for these two attributes respectively then the assumption can be stated as f(xi x yi) = f(xi) x f(yi) for i = 1, 2, ..., n. To split either in the x dimension or in the y dimension on i^{th} slice the partition will divide the chosen slice into two equal slices, here equality is measured by the approximate number of data items in each slice. In essence, the method actually transforms a non-uniformly distributed data space into a uniform one by splitting the slice on a carefully chosen position. For instance, in a 2-d data space, if the slice sxi in the x dimension is to be split and the total number of data items in sxi is N(xi), then a split will divide the slice into two slices, say, $sxi^{(1)}$ and $sxi^{(2)}$, and the split results in $N(sxi^{(1)}) \approx N(sxi^{(2)})$. The position which the chosen slice will be split is determined by the following method.

Suppose we have chosen sxi in the x^{th} dimension to split and there are sy slices in the y^{th} dimension. To choose where to split we need to know f(xi), the data distribution within slice sxi in the x^{th} dimension. To find the split position:

(i) calculate the total number of the data items in the i^{th} slice |N(sxi)| and half its value:

$$\left\lfloor \frac{N(sxi)}{2} \right\rfloor$$

(ii) calculate signatures for all grid cells in sxi: z(xi, y) for y = 1, 2, ..., sy;

(iii) order all grid cells on values of attribute x for all data items stored in sxi:

$\{d1, d2, ..., dk \mid x1 \leq x2 \leq ... \leq xi\}$;

(iv) identify the middle data item in the sorted data set:

$d_{\lfloor N(sxi)/2 \rfloor}$ as the position to split slice sxi.

The advantage of this method is that the split is based on a chosen position which transforms a non-uniformly distributed data space into a uniformly distributed data space. However, the transformation is very complex, especially when m > 2. For a 2-d space, as in the above example, sy data buckets need to be examined and there are sy x b data items or so that need to be sorted before the decision can be made. A brief explanation of the pattern is shown in Figure 4.6.

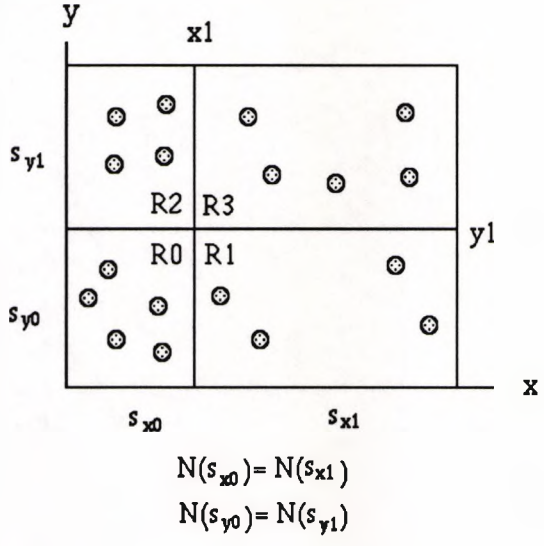
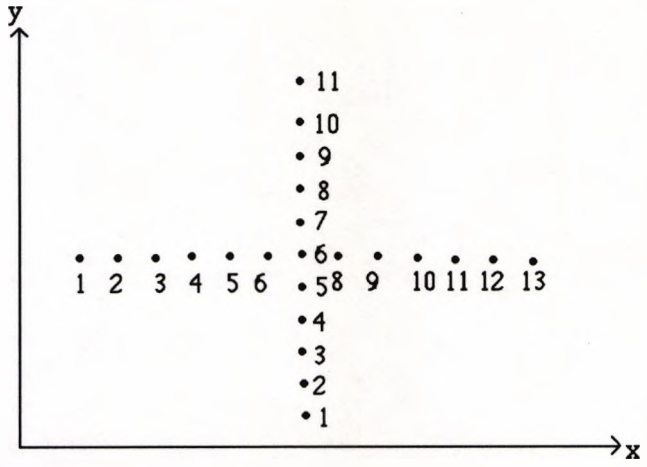


Figure 4.6. Data distribution controlled partition.

PT5 Partitioning the data space by region/brick [LO87] where the embedded regions are allowed (the BANG file or hB-Tree partition)

All the partitions given previously carry out the split along each dimension alternately by a (m-1)-dimensional super-plane. Each split creates new boundaries for a chosen dimension, i.e. for a 2-d search space a split divides the search space by a line; in a 3-d search space a split partitions the search space by a 2-d plane. None of the above partitioning methods can form a m-d region in the search space by a single split.

The partition type PT5 divides the data space into bricks, Viz the super-rectangle regions. One split will create a m-d rectangle region in the data space. This method can partition the search space for a specific data distribution pattern which the other partition methods cannot deal with effectively. A typical example for this kind of distribution is shown in Figure 4.7



X = x1x2x3x4x5 (in binary)
Y = y1y2y3y4y5
z = x1y1x2y2x3y3x4y4x5y5

For the give figures in the diagram we have the following z-code for those points shown ((7, 1 - 11), (1 - 13, 6))

z(7,1) = 00101011	z(1,6) = 00010110	z(7,1) = 00101011	z(1,6) = 00011110
z(7,2) = 00101110	z(2,6) = 00011100	z(7,2) = 00101110	z(2,6) = 00011100
z(7,3) = 00101111	z(3,6) = 00011110	z(7,3) = 00101111	z(3,6) = 00011110
z(7,4) = 00111010	z(4,6) = 00110100	z(7,4) = 00111010	z(4,6) = 00110100
z(7,5) = 00111011	z(5,6) = 00110110	z(7,5) = 00111011	z(5,6) = 00110110
z(7,6) = 00111110	z(6,6) = 00111100	z(7,6) = 00111110	z(6,6) = 00111100
z(7,7) = 00111111	z(7,6) = 00111110	z(7,7) = 00111111	z(7,6) = 00111110
z(7,8) = 01101010	z(8,6) = 10010100	z(7,8) = 01101010	z(8,6) = 10010100
z(7,9) = 01101011	z(9,6) = 10010110	z(7,9) = 01101011	z(9,6) = 10010110
z(7,A) = 01101110	z(A,6) = 10011100	z(7,A) = 01101110	z(A,6) = 10011100
z(7,B) = 01101111	z(B,6) = 10011110	z(7,B) = 01101111	z(B,6) = 10011110
	z(C,6) = 10110100		z(C,6) = 10110100
	z(D,6) = 10110110		z(D,6) = 10110110

Figure 4.7 Special data distribution for a 2-d search space.

When these points are coded the problem of identifying the distribution become pattern recognition. One can see from the coding that this kind of data distribution has certain bits unchanged in their z-codes.

To recognise this data distribution pattern the data space is partitioned into a number of equal sized grid cells and numbered by the z-code. The interpretation of the data pattern can then be obtained by manipulating the values of the z-codes. Meaningful information is extracted to identify this kind of distribution. Figure 4.7. shows the values of the z-code for this data pattern with sixteen grid cells. It can be seen that the pattern is reflected in the z-code values with certain bits unchanged. For such a data pattern, we see that it is difficult to use a line to divide the data space into two balanced regions which contain approximately the same number of data items (points in the diagram). However, using Bang file partitioning we can create two embedded rectangles to divide the data space into two balanced regions. A partitioning process for a BANG file is shown in Figure 4.8.

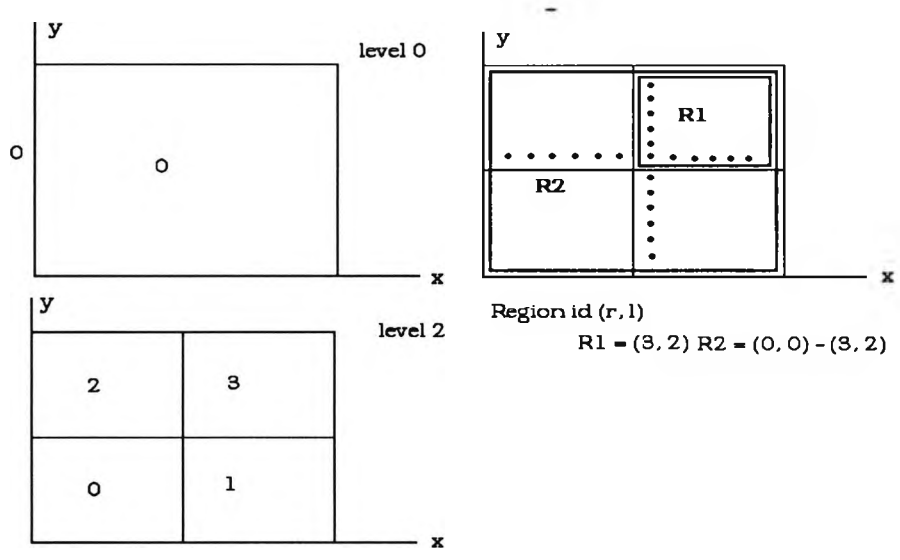


Figure 4.8 Bang File Partitioning for a 2-d Search Space.

(2) The strategies of coping with insertions and deletions

Insertions which cause splitting can be treated as a partitioning process (described above) and deletions can be dealt with as an inverse process to partitioning. When the local data density value becomes low, say, Ld(i,j) is less than required packing density, a merge process is activated which combines two slices into one.

(3) Storage of the access paths

To retrieve data items in each grid cell different methods can be used to store the access paths. We represent these methods as a set: SA = { SA1, SA2, SA3, SA4 }. Each of them is used to implement the access paths of a partition, establishing a relationship between a data item and its address of secondary storage. The meaning of each element in the set is explained by the following diagrams.

SA1 stores a partition by using an index file (see Figure 4.9.).

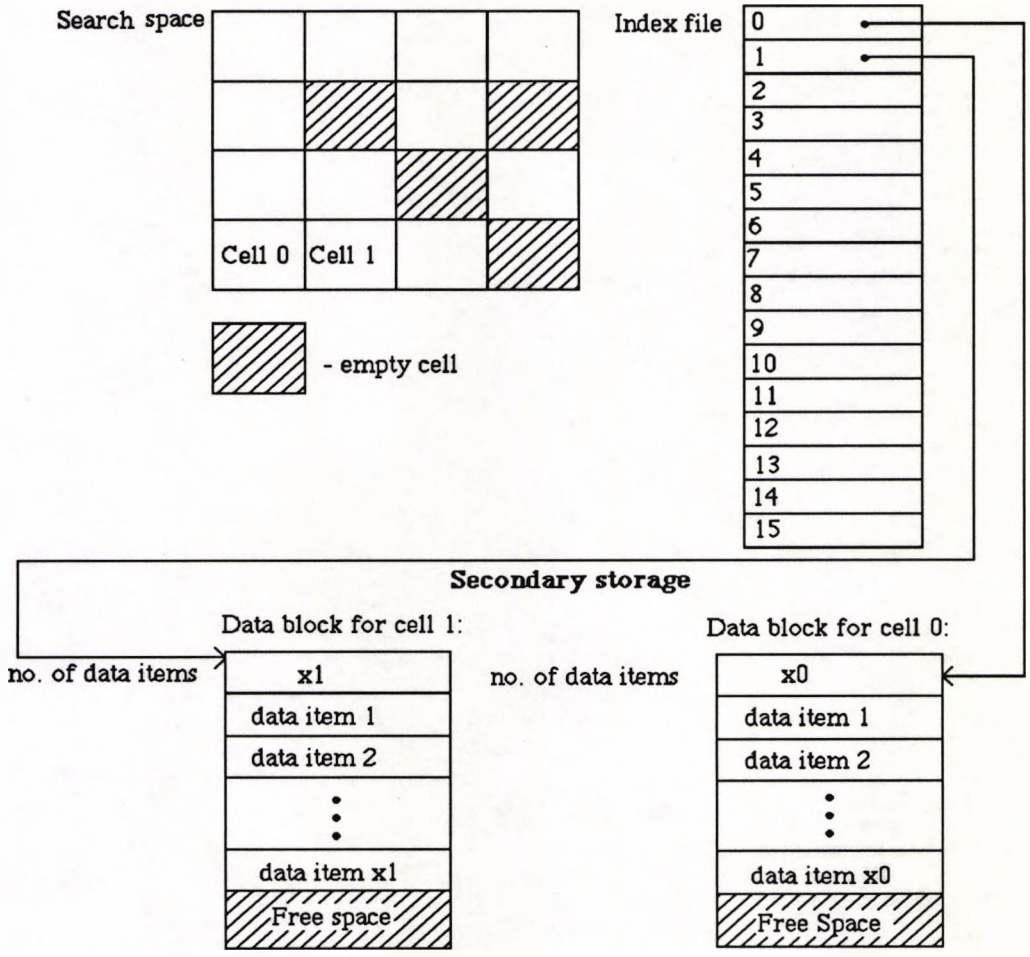


Figure 4.9. Storing the access paths by an indexing file.

In the diagram each non-empty grid cell corresponds to a data block on secondary storage and every grid cell (including empty ones) occupies an entry in the index.

SA2 stores a partition by a hashing algorithm (see Figure 4.10.).

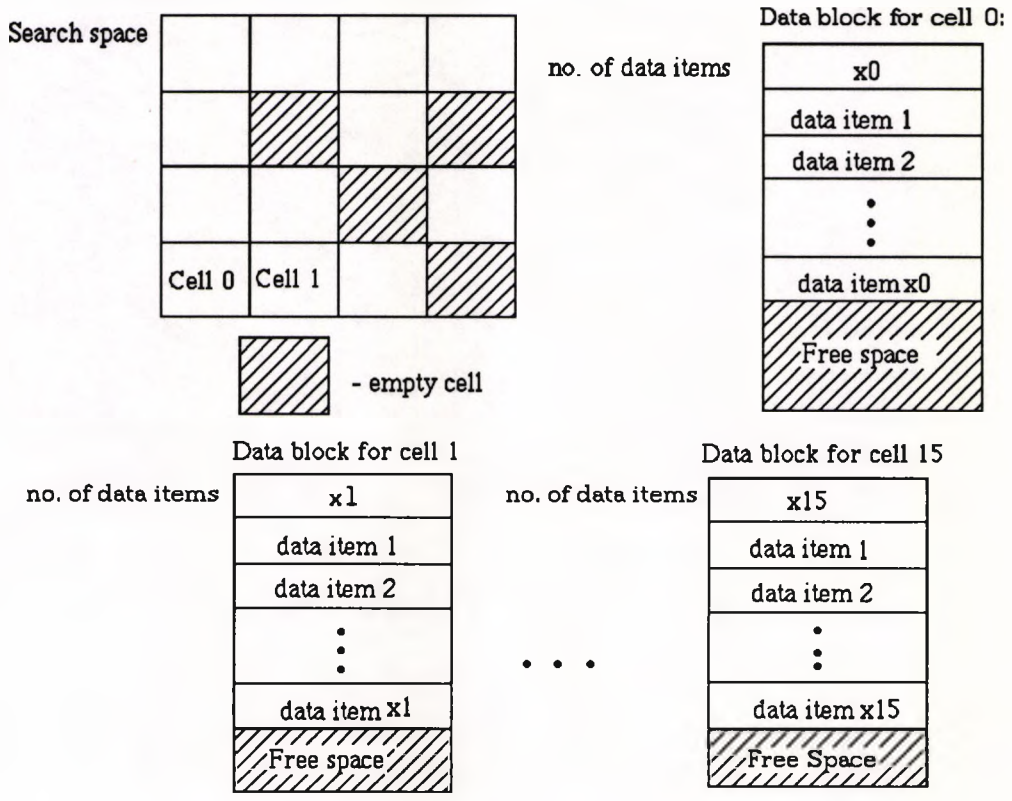


Figure 4.10. Storing the access paths by a hashing function.

In the diagram by applying a hashing function to each grid cell, including empty cells, each cell maps a data block in secondary storage.

SA3 stores PT5 by hB-Tree indexing (see Figure 4.11.).

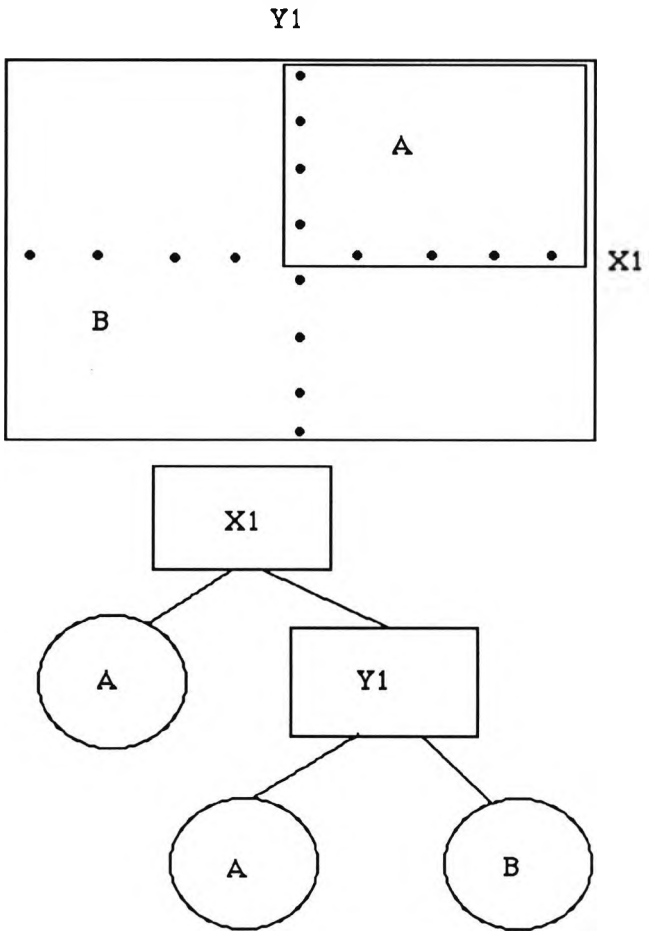


Figure 4.11. Storing the access paths by hB-tree indexing.

SA4 stores PT5 by the BANG file approach (see Figure 4.12.).

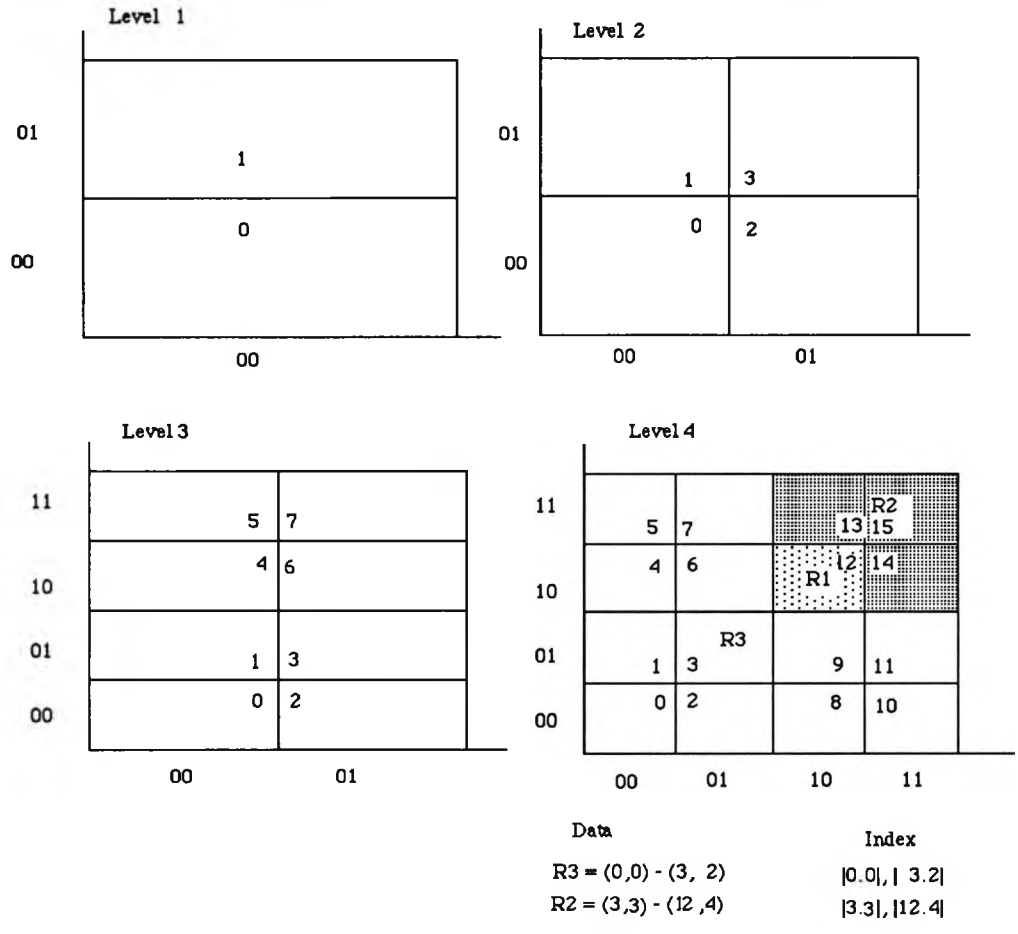


Figure 4.12. Storing the access paths by Bang file approach.

4.2. Various Implementation Algorithms and Their Features

Having extracted the feature information for m-d access algorithms - partitioning and implementing strategies - we can use different combinations of these features to describe an algorithm. For example, the "EXCELL" algorithm can be described by the partition PT1 and access storage SA1 as a pair (PT1, SA1). The grid partition PT1 divides the data space into equal sized grid cells. The storage of access paths (SA1) then implements the partition PT1 by an index file. Each entry of the index file corresponds to a grid cell in the partition and stores the address of the grid cell. The relative position of the index entry is then derived from an array-like calculation. Every grid cell can be uniquely identified by a pair of values (x, y). The index can thus be viewed as a 2-d array. As each grid cell corresponds to an entry in the index file empty grid cells will also have their index entries. Similarly we can describe the z-hashing algorithm as (PT1, SA2). We see that the EXCELL and the z-hashing algorithm use the same partitioning approach, but employ different strategies for storing access paths. The SA2 establishes a relationship between a grid cell and an address for that grid cell through a function, which produces a uniform address for each cell. Different combinations of partitions and storage of access paths can describe different algorithms and therefore, all algorithms can be simplified as algorithm types: ALTi = {partition: PTi, storage of access paths: SAj}, where i is within the range of available partitions and j is within the range of available methods for storage of access paths. For each algorithm type, we will identify which algorithms belong to it and analyse their strengths and weaknesses. We will also find out for which categories of applications each algorithm type is likely to provide better performance. All these judgments are based on current experience, heuristics, performance evaluation, an understanding of the available algorithms in literature and a set of chosen criteria. Further arguments are welcome to improve the selecting algorithms for a tuning process. The system will provide facilities to expand the system knowledge.

We have initially chosen the following algorithms for the model: the EXCELL [TA82], the z-hashing (z-ordering by binary code [HU88a] and z-ordering by Gray code [FA86b]), quantile-hashing [KS87] [KS88b], the PLOP-hashing [KS88a], the hB-Tree [LO87], the BANG file [FR87] and the R-tree [GU84]. Each of these represents a different type of algorithm in terms of partitioning and implementation developed for m-d search spaces. The EXCELL algorithm employs equal sized grid cell partitioning and stores its access paths by an index.The z-hashing also

partitions the data space into equal sized grid cells but implements the access paths by a hash function. The quantile-hashing and the PLOP-hashing use distribution controlled partitionings and implement these partitionings by a hash function. The BANG file divides the data space into m-d super-rectangles and realises the access paths by an indexing approach. The R-tree algorithm is designed for spatial object database and it uses an object-oriented partition implemented by an index approach.

This section describes how an algorithm is chosen if given a set of characteristics, and what the algorithm and heuristic judgments are. Our choice is based on the criteria of a group of characteristics or conditions against the features of a set of algorithm candidates. We study what kinds of application characteristics can match the nature of an algorithm by a comparison of these available algorithms.

4.2.1. The EXCELL Algorithm: ALT[1] = {PT1, SA1}

(1) The characteristics which make the EXCELL algorithm favourable

The EXCELL method is briefly illustrated in Figure 4.13.

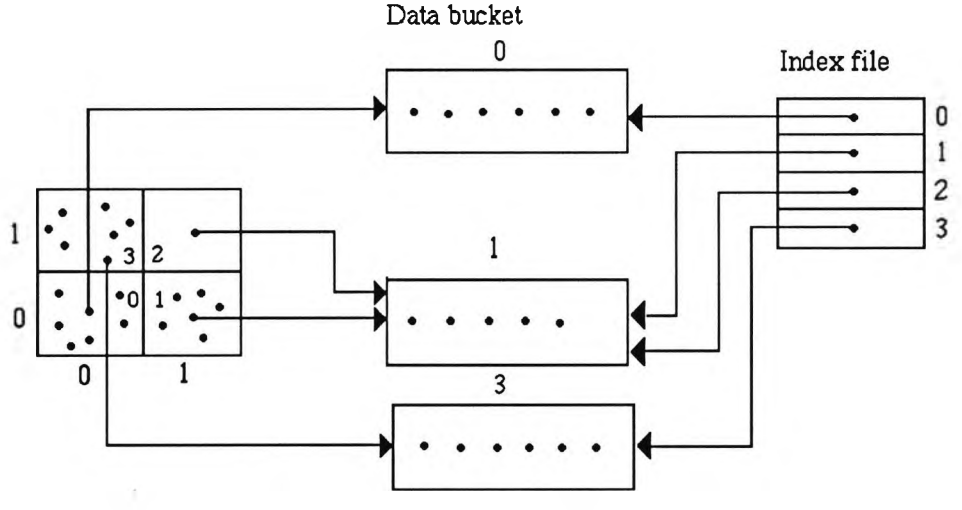


Figure 4.13. The EXCELL algorithm.

In the diagram there are four grid cells in the partition and one of them is an empty one. Three data buckets are allocated for the partition and four index entries are in the index file for the partition, including one for the empty grid cell.

To set up the characteristics we will use Cij to represent conditions or constraints which make a boundary for these characteristics. Combined with each characteristic they form the basis of left hand side (LHS) of a production system.

(a) The number of empty grid cells produced by the partition does not exceed a "certain" limit: C11. The meaning of the word "certain" here relates to environment constraints and tradeoff between the indexing and the hashing algorithms. For instance, an index can be used to avoid data holes in the data file, but it needs storage space and retrieval time. In terms of speed, storing the index in main memory is preferable. If there is enough space in main memory for an entire index file then retrieving a data item through an index is comparable to doing so through a hashing approach, since otherwise an extra secondary storage access will be required. If the storage utilisation is considered and if the space used by an index is less than the space occupied by the data holes of a partition then storage space will be gained by using an index. To decide the limit C11 an estimation calculation can be found in Appendix A2.

(b) The life span of the data set is short: C12. In relational databases, these data sets can be the temporary tables built up by join, projection, and other operations. They can also be intermediate data sets which are only used for a specific purpose. Once the purpose is fulfilled, the life of the data sets are terminated.

(c) The size of data set is "small". Here "small" depends on the environment. It means that the available memory will have enough room to hold the entire index file for the data set of its grid cells (refer to Appendix A2 for its calculations): C13.

(d) The rate of insertions and deletions is moderate: C14. If the index implementation is compared with the hashing implementation and when C11 and C13 are satisfied, it copes with the dynamic situation better than hashing due to the complexity involved in rearranging the data set on secondary storage, i.e. when a split or a merge occurs the indexing method shuffles index records whereas a hashing algorithm shuffles the data

buckets. Since the data set size is, in most cases, greater than the index file size, changing the index file will be more efficient than changing the data set. The way we make a decision on data volatility depends on the speed of performing an insertion or deletion operation as well as the effect on performance. The method for making decisions concerning C14 can be referenced in the Appendix A2.

(e) The response time allows for two secondary storage accesses to retrieve a data item: C15.

To make a choice based on these characteristics, it is preferable to employ the simplest conditions in the reasoning sequence and therefore, we can derive our rules for selecting the EXCELL algorithm as:

Rule set 1 (Rset1)

(R1.1) If C12 and C15 then ALT[1]

(R1.2) If C13 then ALT[1].

(R1.3) If C15 then ALT[1].

(R1.4) If (C11 and C14) then ALT[1] else Rset2.

(2) The reasons for setting these characteristics to choose the EXCELL algorithm are that the ALT[1] algorithm represents the simplest implementation among chosen algorithms. Since its partition divides the data space into equal sized grid cells the scale is fixed for every grid cell, implying an easy transformation between a data identification and corresponding slices. The access paths are implemented by a one-to-one mapping between each grid cell and the relevant index entry, implying that arbitrary addresses can be allocated to a data bucket as the address of the data item is recorded in the index file. Hence it does not require contiguous storage whereas the directoryless hashing algorithms do. We choose the EXCELL algorithm because its implementation is straightforward. When the above mentioned conditions are met, the index file can be stored in main memory or the index size can balance the reserved storage for empty grid cells, the method is compatible with a hashing algorithm in that it provides fast access. In addition, EXCELL copes with a dynamic situation in a simpler manner (index level rather than data level). By keeping an entry in the index for every grid cell, the index can be addressed directly through an equivalent z-code formed from a given data item. The z-code, which is used

as a hashing function to address the index file may also assist VLR (variable length record) data items because the index records have fixed lengths that facilitate applying a hash function.

(3) The strengths and weaknesses of the EXCELL algorithm

The purpose for the analysis of the strengths and weaknesses of each algorithm is to provide the system with knowledge that helps to eliminate unnecessary rule searching and rule matching. As the EXCELL algorithm divides the data space into equal sized grid cells and uses an index to store its access paths, every grid cell has an entry in the index file recording the addresses of data items stored in that grid cell. When searching a data item, the multi-keys are used to calculate the entry to the index file and then the data item is located by the content of address field in this index entry. The algorithm is pictured as in Figure 4.9. Conditions under which the algorithm is particularly strong or weak are listed below.

(a) Equal sized grid cell partitioning makes implementation simple, implying that with a data set which is small and has a short life span it will always be a good choice. The reason is that a small data set will automatically limit the size of the index.

(b) As the index file records the addresses of data, data items of various lengths can be handled.

(c) Every grid cell has an entry in the index, indicating that an increasing number of empty grid cells will increase the depth of the index and influence the search speed. On the other hand, it also removes the holes in the storage of the data set itself. Hence it can guarantee good performance for applications with a small data set. Small sized data sets naturally limit the height of the index even for a non-uniform data distribution.

(d) Speed and storage utilisation depends on the data distribution. Namely, the algorithm is dependent on the size of the data set and data distributions.

(e) When the index file cannot be held in main memory the access speed will be influenced, but only by one extra secondary disk access for point data because the index record has a fixed record size and therefore only one probe by a hashing function is required to locate the index.

(f) When there are very few holes in the file the index file will not be necessary, since empty holes will occupy less space than an index file. This implies that for evenly-distributed data sets hashing will perform better than EXCELL if in addition, the insertion and deletion rate are not very high.

4.2.2. The z-hashing Algorithm: ALT[2] = {PT1, SA2}

The binary code z-hashing algorithm ALT[2.1]
The Gray code z-hashing algorithm ALT[2.2]

ALT[2] represents the z-hashing algorithms. Two types of z-hashing are considered here. One is the binary code ordered z-hashing ALT[2.1] (see Figure 4.14.(a)), the other is the Gray code ordered z-hashing ALT[2.2] (see Figure 4.14. (b)). When using the same partition strategy as the EXCELL, the hashing algorithm, instead of mapping each grid cell to an entry address referring to an index file, creates a data bucket number (a relative address on secondary storage) which corresponds to each grid cell.

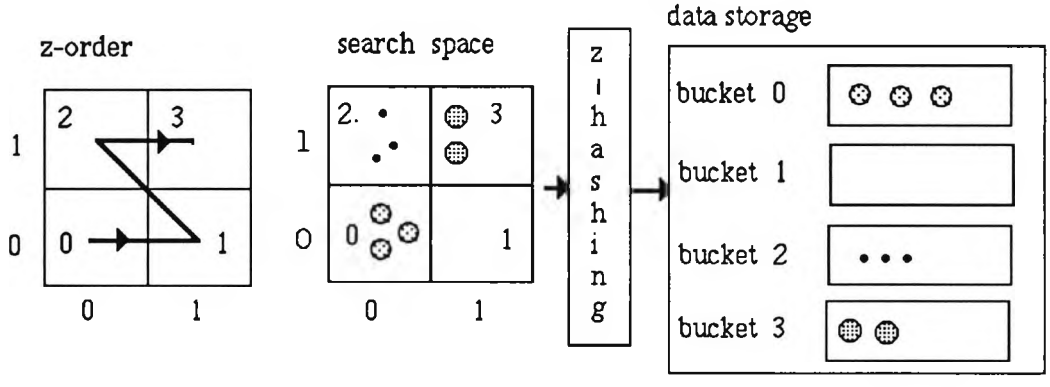


Figure 4.14.(a) Binary code z-hashing algorithm.

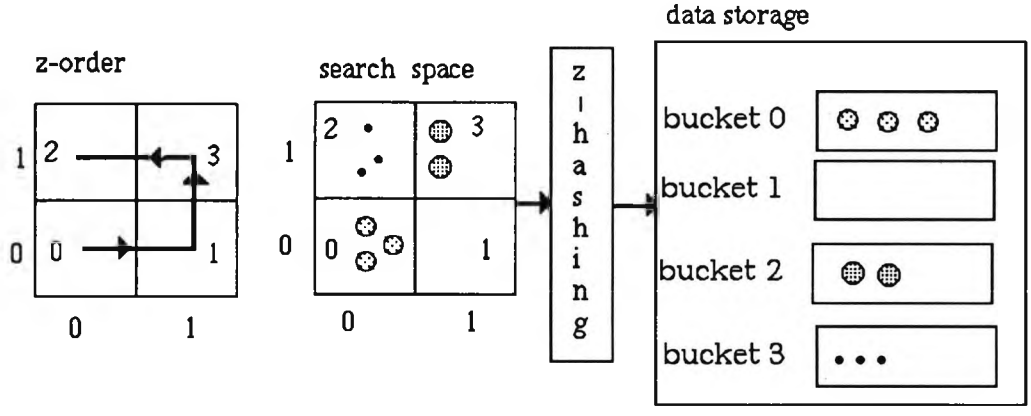


Figure 4.14.(b) Gray code z-hashing algorithm.

The binary code z-hashing algorithm ALT[2.1] divides the data space into equal sized grid cells and numbers these grid cells in the z-order. In addition, it tries to optimise performance by choosing the order in which grid cells will be split or merged next when dealing with dynamic situations. The idea is that the grid cells chosen will minimise the number of data buckets to be utilised. This is done by selecting the grid cells that produce the lowest z-code values among the unused bucket numbers. As the hashing function generates the bucket address number by mapping a chosen grid cell to a z-code, a split will introduce a new bucket number sequence (2^L, $2^L + 1$, ..., $2^L + 1$), where L is the data set level before a split. Hence the split rule will select those grid cells which introduce the bucket numbers of 2^L, $2^L + 1$ as the first one to split. In a partition each dimension is divided into a number of slices. Every slice in the i^{th} dimension is numbered by 0, 1, ..., s_{i-1}, where s_i is the number of slices in the i^{th} dimension. Each grid cell can thus be identified by these slice numbers as $[i_1, i_2 ..., i_m]$, where i_j is the $(i+1)^{th}$ slice on the j^{th} dimension for i = 0, 1, ..., s_{i-1}, j = 1, ..., m. The z-code of a grid cell identified by slices $[i_1, i_2 ..., i_m]$ is generated by interleaving the

binary representation of each element in $[i_1, i_2, \ldots, i_m]$ alternately. The z-code then serves as an address (a bucket number) for the corresponding grid cell. A one-to-one relationship between a grid cell and a bucket number is established by the z-hashing function. From the current identification of slices in every dimension the grid cell that will produce the lowest new bucket number can be calculated by an inverse function, i.e. a function which transfers the lowest new bucket number into its corresponding grid cell slice identification $[i_1, i_2 \ldots, i_m]$. By the splitting rule, the number of gaps among the buckets is minimised. An explicit illustration of the split rule for the process is illustrated in Appendix A3.

The special feature of z-hashing is that the splitting and merging processes aim at minimising the number of empty buckets so that the storage utilisation may be improved and the number of grid cells requiring to be rehashed may also be reduced. When the insertion pattern matches the order of the split rule the overall performance will reach the expected results. However, if the insertion pattern does not match the order set by the split rule then empty buckets cannot be avoided so the method will not perform as well as expected. Figure 2. in Appendix A3 shows a data space that originally had 16 grid cells and the corresponding changes of z-codes brought about by using the minimal numbering split rule.

To be aware of the type kind of insertion pattern that can be categorised as suitable for the z-hashing split rule, in terms of the expected performance, the order of insertions plays an important role. An analysis of the insertion pattern for this purpose can also be found in Appendix A3. The analysis will be especially useful when insertions are processed in a cumulative manner, such as a batch process. These data items can be sorted before they are added to the data set.

The Gray code z-hashing ALT[2.2] also utilises z-ordering to partition the data space. The difference lies in that it uses the Gray code to number slices in each dimension of the data space. The Gray code z-ordering is specially developed for heavy range searching problems [LA78]. The binary z-order transforms a m-d data space to a 1-d data space and preserves the geometric proximity locally in a Z shape (see Figure 4.15. (b)). Gray code explores the essence between the binary code and the geometric proximity, regarding the storage of the m-d data, by redefining the sequence of the binary codes. It reorders the binary code in a way such that that only one bit will be different from the next binary code followed. Ordering in this

manner, the Gray code minimises the Hamming distance (the number of bits different in the sequence of codes) and increases the data similarity between the consecutive buckets. Data items with the same attribute values will more likely be stored together. For a partial range search, if each bit stands for an attribute value, then query operations for example, with a search pattern such as ??1? will require all data with the third position of value 1 to be stored as closely as possible. For a binary sequence and a Gray code sequence a set of 4-bits codes is shown in Figure 4.16.

(a) Binary z-code

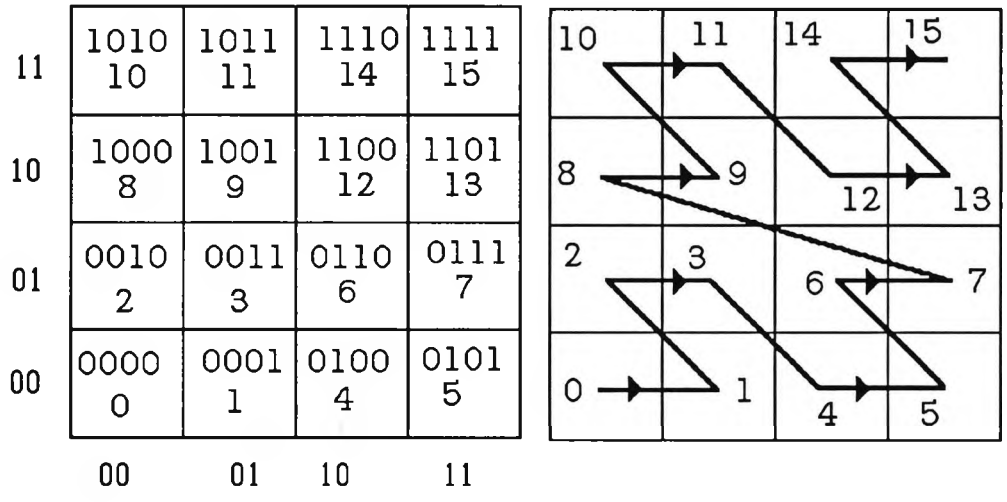
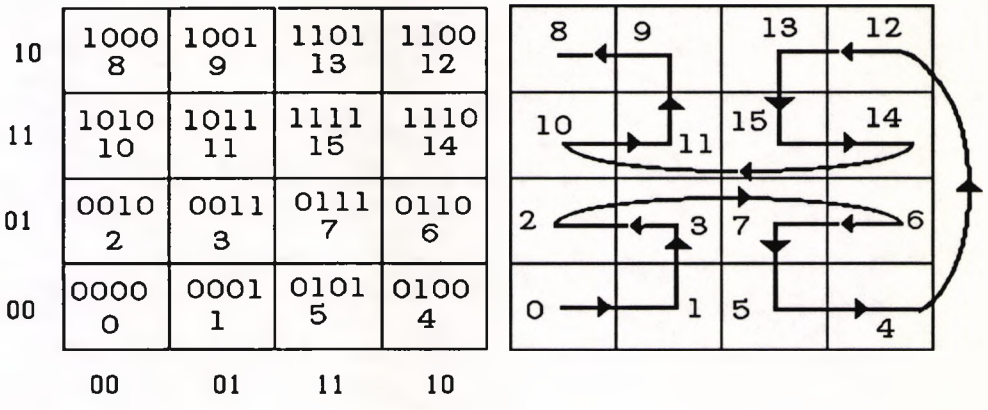


Figure 4.15. Comparison between binary z-code and Gray z-code.

(b) Gray z-code



Binary code	**Gray code**
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0100
0110	0101
0111	0111
1000	1111
1001	1110
1010	1101
1011	1001
1100	1000
1101	1100
1110	1110
1111	1010

Figure 4.16. Binary code and Gray code.

In a binary code sequence there are four discrete sections, whilst in Gray code sequences there are three discrete sections when the third position is 1. As a result, the Gray code achieves more efficient ordering than the binary code for data similarity. The effect of such reordering will improve the performance for range and partial range searching. An example is shown above in Figure 4.15. According to the Gray code z-ordering, the following grid cell only differs by one bit so that there are three bits matched. For the same partition, the binary code z-order will have two bits different in the following grid cell in the diagonal direction. The comparison of these two kinds of z-ordering has been illustrated in Figure 4.15 (a) (b) respectively. The Z shape of the binary z-ordering has become the |_ and _| shape in the Gray code z-ordering.

Compared with the binary code z-ordering, the Gray code z-ordering changes the sequence for numbering the grid cells. As a result, it improves the proximity for range and partial range match accesses at the expense of transformation between a binary code and its equivalent Gray code. The Gray code z-ordering is more complex to implement, but it has all the strengths of the z-hashing and, in addition, it performs more efficiently for range and partial range searches than that of the binary one. Gray code provides an alternative to z-hashing algorithm where partial and full range searches are major factors in design considerations.

(1) Cases where the z-hashing algorithm is favourable

(a) The data set is large, implying that there is not enough memory for a whole index file : C21.
(refer to Appendix A2 to decide C21).

(b) The application requires a quick response for point search : C22.
(C22 = Tsec).

(c) The data distribution is relatively even. Here "even" means that the number of data items in each grid cell will be roughly equal (refer to Appendix A2 for calibration) : C23.

(d) The order of the insertions matches the order of splitting or the insert and delete rate is low : C24.
By " the order of insertions matches the order of splitting " above we mean that the data items to be inserted will fit in the range of corresponding expanded grid cells which are chosen by the splitting rule. Note that in z-ordering, each grid cell represents a fixed region of the data space (i.e. z-

hashing function is a function of arguments: size and position in a search space) and therefore, if the chosen expanded grid cells are always those which correspond to the values of the lowest z-code among the unused ones after a split, there is a possibility of no data items being added which fall in the range of those newly created grid cells. This will result in empty grid cells so that the original objective of the z-hashing scheme of minimising the number of buckets required cannot be achieved. This observation of the order match indicates that the data distribution pattern favoured for using this method will relate to the order of the expansion. Based on the split rule, a sequence of grid cells can be calculated and compared with the order of the insertions. The detailed illustration of the order match shall be given in the Appendix A3.

(e) The range and partial range search rate and the partial range query rate are high :C25.
The meaning of "high" is given in Appendix A2.

(f) The required data packing density is relatively low (eg. < 65%) : C26.
This condition is determined in combination with other conditions.

(g) The required data packing density is relatively high (eg. > 80%) C27.
This condition is determined in combination with other conditions.

Rule set 2 (Rset2)

(R2.1) If C25 then ALT[2.2].
(R2.2) If (C21 and C23) then ATL[2.1].
(R2.3) If C22 then ATL[2.1].
(R2.4) If (C23 and C26) then ATL[2.1].
(R2.5) If (C24 and C27) then ATL[2.1].
(R2.6) If C26 then ALT[2.1] else Rset3.

(2) The reasons why these characteristics match the z-hashing algorithm

One of the reasons for choosing the z-hashing approach is that it provides good performance in terms of retrieval speed. Thus, in a situation where the requirement of response time is difficult to guarantee with the EXCELL (ALT[1]) those rules of selecting the z-hashing algorithms can be examined. The Gray code z-hashing algorithm preserves better spatial proximity. This implies that the range and partial range search performance may outperform ALT[2.1]. With z-hashing the storage utilisation may also be improved compared with ALT[1] because the z-hashing considers the order for the next

grid cell to be split; in particular when the sequence of inserting data items matches the splitting sequence, the z-hashing makes best use of the secondary storage and gives a fast response.

(3) The strengths and weaknesses of the z-hashing algorithms

(a) Storage utilisation is dependent upon the data distribution and the sequence of insertion. Some data distributions will perform well, some may result in a large number of data holes.

(b) Using the z-code to number the grid cells preserves geometric proximity for the data set.

(c) A hashing approach usually provides a fast search speed.

(d) The z-hashing algorithm maps a grid cell to a bucket number for data items within the range of the grid cell, implying that a split may create empty grid cells. The result is possible low storage utilisation and a large amount of data reorganisation.

To compare the EXCELL scheme and z-hashing algorithm, the former uses a hashing function to locate the index file whereas the latter uses a hashing function to locate a data item in a data bucket. The different decisions made between ALT[1] and ALT[2] lie in the required response time and the number of empty grid cells produced by the chosen partition. Thus if PT1 is chosen then selection between ALT[1] and ALT[2] will be based on two factors: (a) the main access mode; (b) the storage utilisation. When the main access mode is real-time (random access is the dominant access mode) ALT[2] may be preferable; otherwise when the number of empty grid cells multiplied by the size of data bucket which exceeds the size of the index file, ALT[1] will be favourable.

In sections 4.2.1. and 4.2.2. we have examined two implementation algorithms with the same type of partitioning. There is a break-even point for choosing one of these two implementations in terms of storage and speed. This has been described by limits C11.

4.2.3. The Quantile-hashing Algorithm ALT[3] = {PT3, SA2} {PT4, SA2}

The quantile-hashing algorithm [KS87] [KS88b] uses binary trees in each dimension to aid the implementation of the hashing scheme. An illustration of quantile-hashing is shown in Figure 4.17. These binary trees are used to keep track of the boundaries of dividing slices (quantiles) for each dimension. All branches of the binary trees are regarded as a digital tree to carry the values of 0s on the left hand side and 1s on the right hand side branches. These 0s and 1s from the root of the tree to the leaf of the tree form a binary string which uniquely represents the relevant slice in that dimension. Slices in every dimension divide the data space into super-rectangles. A super-rectangle is bounded by m slices. Each super-rectangle can thus be identified by these binary codes (in binary strings) which correspond to it. These binary codes then form the basis for a hashing function to calculate the bucket addresses for corresponding grid cells. Other than that the quantiles (splitting positions) can be decided by control functions (i.e. frequency / density / data distribution), in essence, the hashing function is similar to the one used by z-hashing algorithm ALT[2], it only differs in partitioning search space.

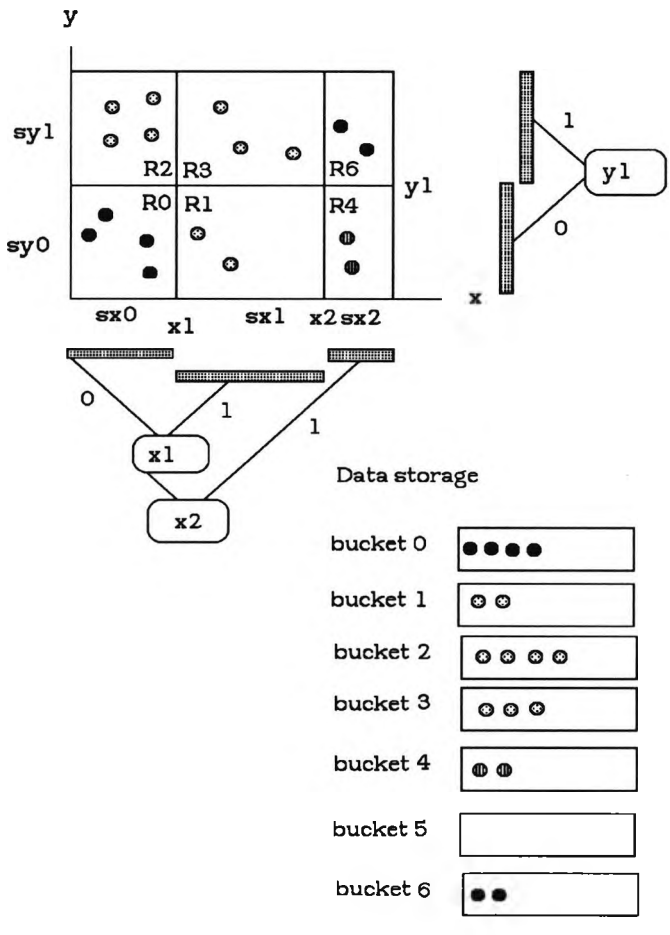


Figure 4.17. Quantile-hashing algorithm.

The idea of the quantile-hashing is to apply a number of very small* indices to support its hashing function, which allows control over the splitting process and takes non-uniform data distribution into consideration. As the boundaries of slices and local density values are kept in the binary trees the algorithm makes it possible to vary the sizes of grid cells and allows various split frequencies for different dimensions. However, every time an expansion occurs a split will cause a reorganisation of the data file for these affected regions, especially when $m > 2$. During a splitting process, the new addresses are calculated by a G-function [OT84], which organises grid cells in lexicographical order unlike the z-ordering. To achieve easy implementation and better geometric proximity the G-function can be replaced by the z-ordering function.

(1) Cases where the quantile-hashing algorithm is favourable

(a) The required resolution is too high for a given bucket size so that it creates many empty grid cells due to equal-sized grid cell partition: C31.
The reason is that the data distribution is non-uniform and thus the partition may create too many empty grid cells. As a result, the z-hashing method cannot cope with non-uniform data distribution efficiently.

(b) Fast response time for point search: C22.

(c) Insertion and deletion rate is low: C32.

(d) The range or partial range search rate is high: C25.

(e) The data set is large: C21.
To preserve geometric proximity, we will assume that the address calculated in the quantile-hashing is produced by z-code transformation instead of the G-function [OT84]. This assumption simplifies the DBMS for physical DB organisation because the z-code and G-function would be required to be implemented and replacing the G-function reduces the number of functions supported by the system.

---

* small indices: a grid index needs $\pi_{i=1}^{m} s_i$ entries in an index, whereas a quantile-hashing requires $\sum_{i=1}^{m} s_i$ entries in an index.

Rule set 3 (Rset3)

(R3.1) If (C31 and C22) then ALT[3].

(R3.2) If (C31 and C32) then ALT[3].

(R3.3) If (C31 and C25) then ALT[3].

(R3.4) If (C21 and C32) then ALT[3].

(2) The reasons for setting these characteristics to choose the quantile-hashing algorithm

ALT[3] offers flexibility for different splitting strategies. By recording the boundaries of these slices and local densities in binary trees for each dimension, it copes with non-uniform data distribution better than the previous algorithms. It also preserves the geometric proximity of z-ordering as the binary trees form the same numbering scheme as z-hashing.

(3) The strengths and weaknesses of the quantile-hashing algorithm

(a) Control can be introduced in the splitting process thus providing the flexibility of partitioning the data space.

(b) The geometric proximity of the z-ordering is preserved.

(c) Reorganising the data space for a split process is required.

(d) Each search has to consult these binary trees first before locating the required data items and the calculation of data address is more complicated than equal sized grid cell partition. When the number of dimensions is small, all binary trees can be easily stored in main memory. Otherwise, consulting these trees may have extra overhead.

(e) Quantile-hashing is a hashing algorithm such that there are holes in the data file for empty cells produced by the partition. Since the partition allows varied size, compared with EXCELL it reduces the number of empty grid cells.

4.2.4. PLOP-hashing Algorithm: ALT[3] = {PT3, SA2}

All the above mentioned implementation algorithms require a relatively large amount of data reorganisation when a split takes place. The reason is that with the z-hashing algorithm mapping both grid size and its position to a bucket number when a split occurs the resolution will change, implying the size of these grid cells will also change. This is undesirable for a highly dynamic data set. The PLOP-hashing tries

to overcome this problem by introducing a dynamic numbering scheme to these grid cells.

The PLOP-hashing algorithm, like the quantile-hashing, uses binary trees for each dimension to record local data densities and boundaries of slices; but, unlike the quantile-hashing, the identification of each slice is not formed by defaulted 0s and 1s on the paths of these binary trees. Each number for a slice identification is **dynamically** produced and recorded in the leaves of these binary tree indices. Hence it improves the ability to cope with a dynamic situation. Moreover, rehashing is also localised to the affected areas for each split and merge. Non-uniform data distribution can be considered as in the quantile-hashing so that it is not necessary for a split to be a middle point in the data space for a split grid cell. However, to gain these properties extra information about dynamically formed slice numbers needs to be recorded in these binary trees. Comparing PLOP-hashing with the z-hashing approach these extra binary trees are required whilst comparing PLOP-hashing with quantile-hashing the tree size will be increased. A brief illustration is shown in Figure 4.18.

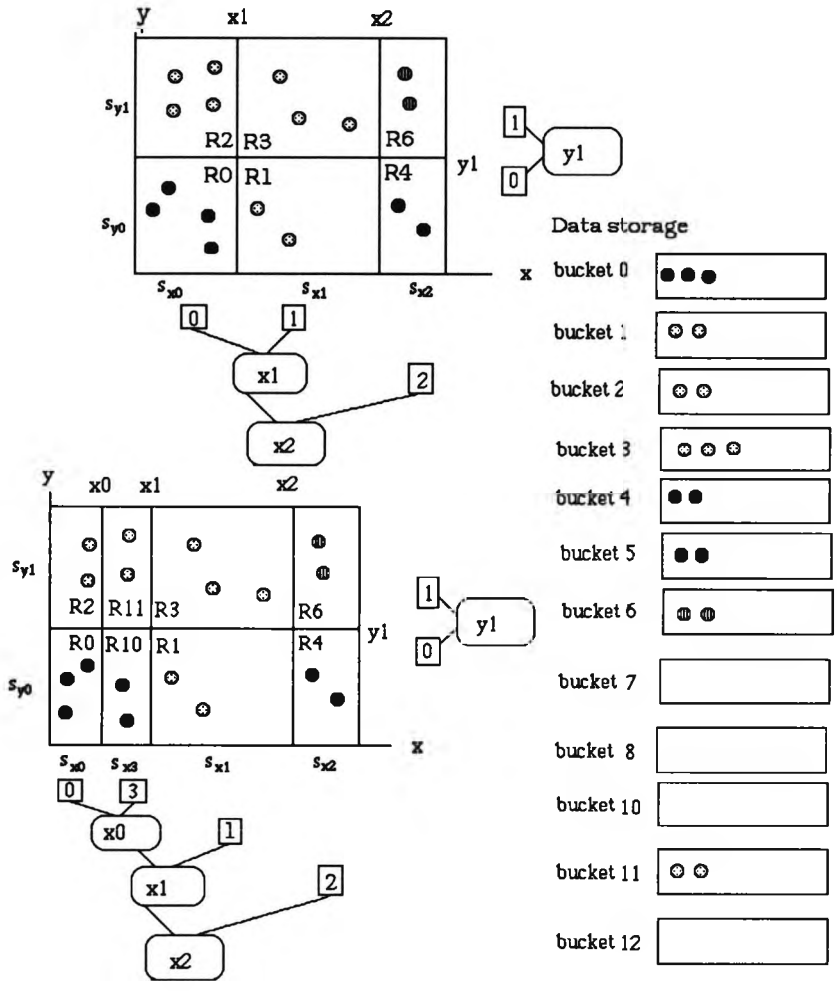


Figure 4.18. The PLOP-hashing algorithm.

The difference between the PLOP hashing and the z-hashing algorithm is that the PLOP-hashing can apply different scales to divide the data space and different dimensions can have a different number of slices. The unnecessary partition can be avoided. In addition, unlike the z-hashing and the quantile-hashing, the sequence of address calculation in the PLOP-hashing is dynamic in correspondence with the sequence of data growth so that the identification numbers given to a grid cell are not fixed. This can be shown by Figure 4.19. This feature implies that the indices used for address calculation are adapted for dynamic situations, thus the data reorganisation caused by a split is localised. Although it copes with dynamic situations better than other algorithms because less grid cells need to be rehashed during the growths or shrinks of the data set, the geometric proximity may not be preserved as the indices are formed dynamically and do not necessarily follow the expected z-order for neighbouring grid cells. As a result, it may give a slow response time for range searches.

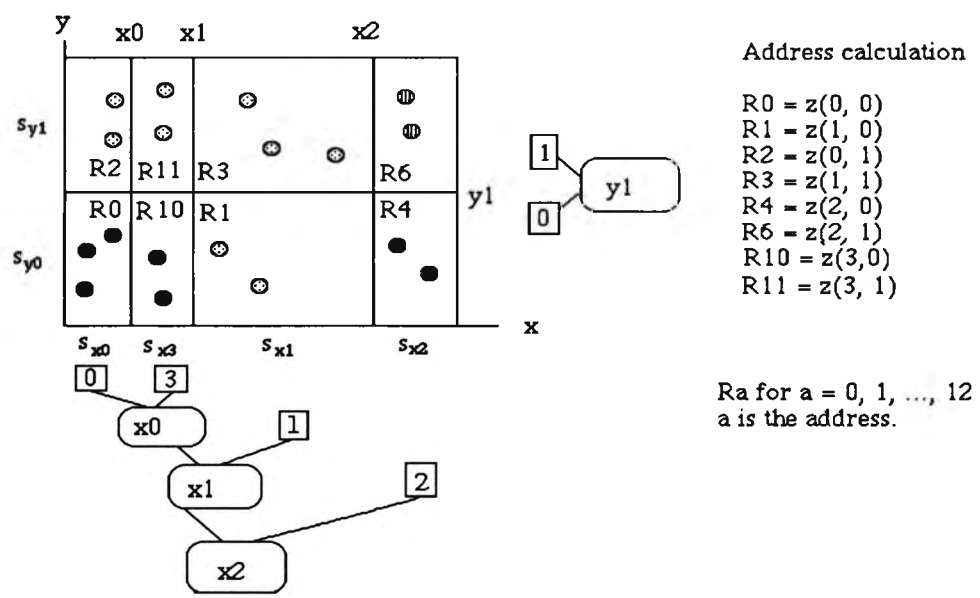


Figure 4.19. The PLOP-hashing address calculation.

(1) Cases where the PLOP-hashing algorithm is preferable

(a) The data distribution is non-uniform : C41.

(b) The fast response time for point search is required : C22.

(c) The data set is large : C21.

(d) The insertion and deletion rates are high and unpredictable which requires the changing of resolution accordingly : C42.

(e) Range and partial range search rates are low : C43.

(f) There are large differences among local data densities : C44.

(g) Searches for attributes are at different rates : C45.

Rule set 4 (Rset4)

(R4.1) If (C22 and C43) then ALT[4].

(R4.2) If C41 then ALT[4].

(R4.3) If C43 then ALT[4].

(R4.4) If C44 then ALT[4].

(R4.5) If (C41 and C45) then ALT[4].

(2) The reasons for setting these characteristics to choose the PLOP-hashing algorithm

The ALT[4] is a fast search scheme implemented by a hashing algorithm employing G-ordering. It provides fast access supported by the hashing algorithm and small size binary trees. It adapts different dividing intervals (resolution of the data space) so that it copes with dynamic situations efficiently. It supports PT3 and PT4 types of partition strategies. Hence a non-uniform data distribution may be transformed into an uniform distribution by PT4 (non-equal sized grid cell partition). The PLOP-hashing is selected mainly for its ability to deal with dynamic situations and its effective point retrieval.

(3) The strengths and weaknesses of the PLOP-hashing algorithm

(a) The PLOP-hashing algorithm provides flexible splitting and merging by recording indices in the binary trees.

(b) Fast response is achieved by a hashing function.

(c) Dynamic situations are dealt with effectively by an adapted numbering scheme for slices.

(d) The PLOP-hashing algorithm may lose geometric proximity as z-order is

applied to insertion sequence rather than geometric data space.

4.2.5. BANG File and hB-tree Algorithm: ALT[5] = {PT4, SA1}

With reference to the partition type in Figure 4.7. we know that not all of the above mentioned algorithms can cope with specific data distribution efficiently. As shown in Figure 4.20., even applying the PLOP-hashing partition, which is developed to deal with non-uniform distribution, many empty buckets may still be introduced for this specific data pattern. Moreover, these algorithms cannot deal with overflow effectively. This is because each grid cell corresponds to a data bucket and, if the number of data items exceeds the size of the bucket, then it will either trigger a split or introduce an overflow area. As a result, if a split is triggered a number of new grid cells are created and the storage utilisation may become poor; if an overflow area is introduced then to retrieve data items in the overflow area an extra secondary storage access may be required. The Bang file partition divides the data space into m-d regions, avoiding the problem caused by other partitioning strategies.

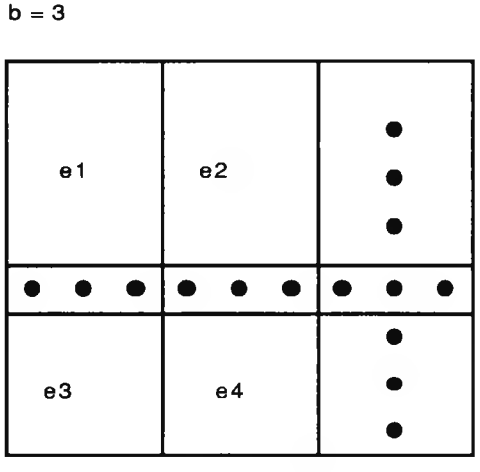


Figure 4.20. Specific data distribution by parameter guided partition.

The figure shows that, among nine grid cells four empty grid cells have managed to occur by PT3.

The BANG file is an interpolation-based grid partition which identifies grid cells of different sizes by different data set levels. The technique used for implementation has to record the data set level and thus the indexing approach has to be employed for this partition. A grid cell is represented by a region identifier and a data set level label pair (r, l). The identifier is formed by concatenating the least significant bit of the newly-formed coordinate in dimension i at level l + 1 to the most significant bit of the corresponding region number at the level l [FR87a] [FR87b]. If a region encloses the other region then the embedded region will be the lowest level in the index. The region which has the higher level number will be searched after the lower level regions. This property guarantees that the higher level region equals the region identified by r minus those lower level regions embedded within it. This can be shown in Figure 4.21.

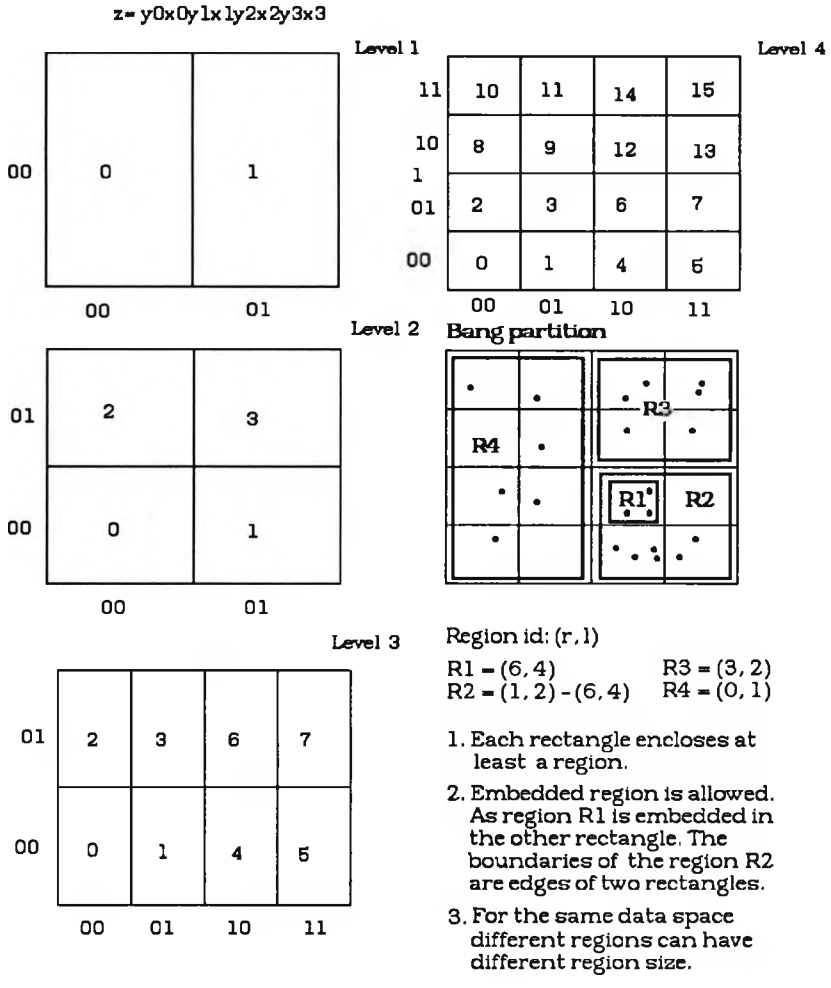


Figure 4.21. The Bang file partition and representation.

The BANG file partition [FR87a] and hB-Tree partition [LO87] are essentially the same. An embedded region in the Bang file is equivalent to a holey brick (bricks with holes in) in a hB-Tree. They both minimise the problems introduced by non-uniform data distribution. The difference between them is the access paths implementation. The hB-tree represents the partition by recording the boundaries of the bricks (regions) at different levels of a k-d-tree alternately; whereas the Bang file represents the partition by its region identifier and a data set level, which gives the location and size of a data item in the data space. The salient feature of this partition is that the dimension of a split equals the dimension of the data space so that it can compact non-uniform data distribution effectively. This feature leads to an implementation advantage: the growth rate of the index size will be at the growth rate of data. It also copes with the dynamic situation at high packing density by introducing no empty regions. Both the region identifier and the data set level are required to identify a grid region. In addition, to gain a better packing density it needs to keep extra information in each index record compared with the scale-based grid indexing method. The extra information is necessary for recording the data set level. This method may make range and partial range searches complicated and slow. The reason is that a data range involving multi-regions may demand the traversal of several branches in the Bang indexing tree. Comparing the the Bang file with the z-hashing algorithm, the Bang file has to use an index file for its access paths so that there will be a time when the break-even point is reached for storage utilisation. The formula for the break even point calculation is presented in Appendix A4.

The BANG file and the hB-Tree implementation were illustrated in Figure 4.11. and Figure 4.12. respectively.

(1) Cases that in favour BANG file algorithm

(a) The range search rate is moderate : C51.

(b) The point search rate is high : C25.

(c) The distribution of data is non-uniform : C41.

(d) The database to be created is region-data-oriented or object-oriented, i.e. is explicit m-d data : C52.

The condition (d) needs rethinking in some situations because for an object in the middle of the data space the signature created by the z-code will fail to code the

object. However, an alteration can be introduced so that the partition can be identified by several layers. The alteration is discussed in section (4) below.

(e) High insertion and deletion rates : C42.

Rule set 5 (Rset5)

(R5.1) If C41 then ALT[5].

(R5.2) If (C51 and C25) then ALT[5].

(R5.3) If C52 then ALT[5].

(R5.4) If (C41 and C42) then ALT[5].

(2) The reasons for setting these characteristics to choose the BANG file algorithm

The ALT[5] is designed to adapt to changes of data distribution. It achieves this objective by avoiding empty grid cells through a partitioning strategy. The adaptability implies that it can cope with dynamic and non-uniform data distribution well. On the other hand, it represents a region by an identifier which indicates the position and size so that it can represent some object data easily. However, to represent all spatial objects the alteration described in section(4) below needs to be introduced.

(3) The strengths and weaknesses of the BANG file algorithm

(a) The Bang file algorithm can cope with various data distributions.

(b) The Bang file algorithm can store certain ranges of spatial objects conveniently.

(c) The Bang file algorithm can provide high storage utilisation by redistributing the partition. An illustration is given in Figure 4.22.

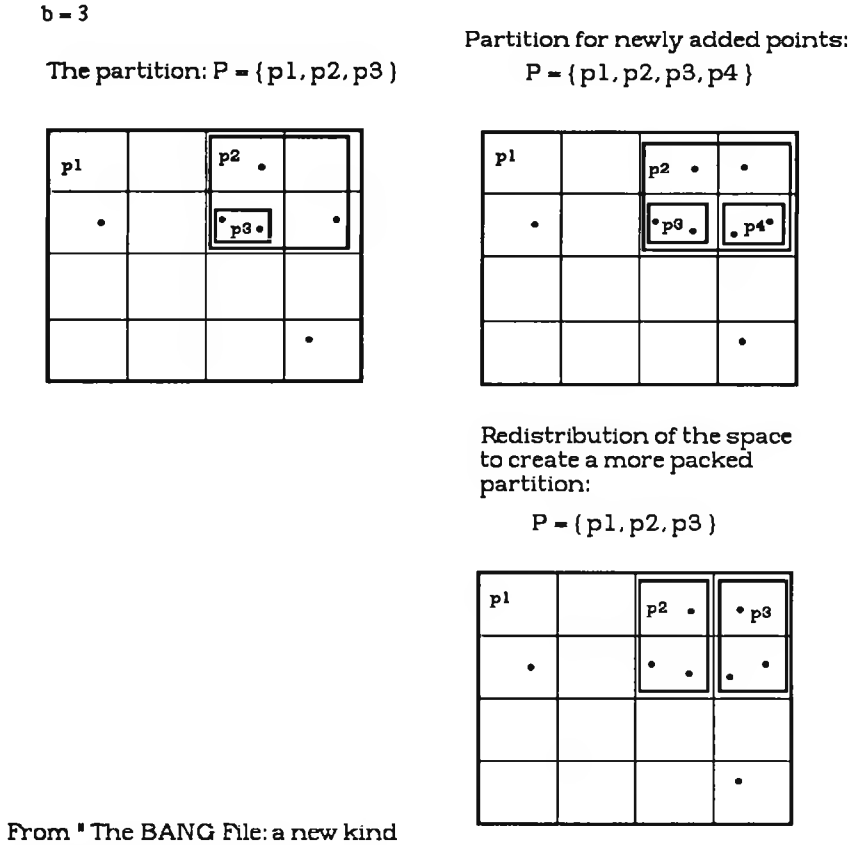


Figure 4.22. Redistribution by the Bang file partition.

(d) The Bang file algorithm can provide fast access to exact search (point search).

(e) The Bang file algorithm may not be very efficient for range and partial range searches.

(f) The Bang file algorithm cannot efficiently store objects which overlap with the partition boundaries without further considerations. (However, all of the above-mentioned algorithms fail to do this).

(g) An index file has to be used to implement the Bang file.

(4) Alterations introduced to BANG file algorithm for spatial databases (development of existing algorithm)

(i) Analysis

The major problem with a m-d spatial object database is that two objects may be positioned in such a way that, for instance, in a 2-d data space, one cannot use a horizontal or vertical line to separate them without dividing one object into two parts. An example is shown in Figure 4.23. it can be seen that line x2 keeps object 3 intact but cuts object 4 into two parts. One layer grid partition, therefore, will fail to identify such an object without cutting it into two or more sub-objects. However, if we imagine that one of the two objects is on the actual plane and the other one is on an imaginary plane, we can apply different lines to these two planes instead of the original one without dividing either of these objects. This method can be seen as a dimension extension approach (another dimension is introduced for identifying different layers of planes, allowing different objects to be mapped on different layers) - a multi-layered approach. Using this approach, whenever there is an object which intersects the boundaries of the grid partition over these existing planes, it will apply an imaginary plane or another layer of plane to avoid dividing the object. Meanwhile, the dividing position is still chosen according to the overall partitioning situation as if there is only one layer plane. An illustration of this method is given in Figure 4.24.

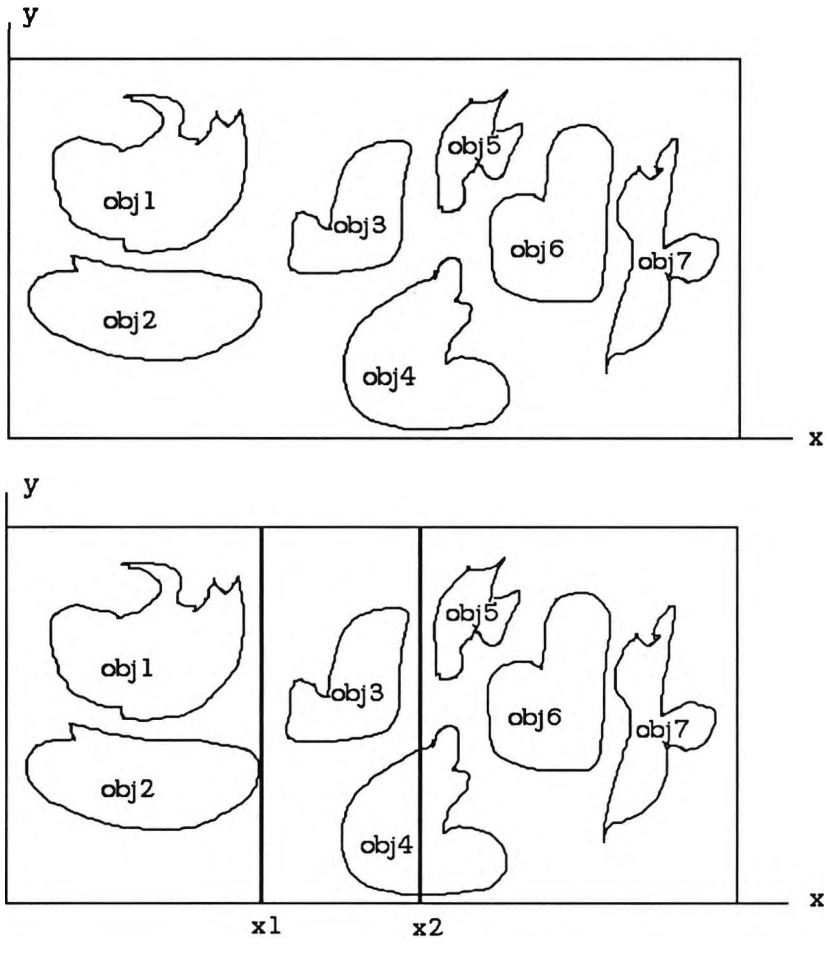


Figure 4.24. Multi-layered 2-d grid partition.

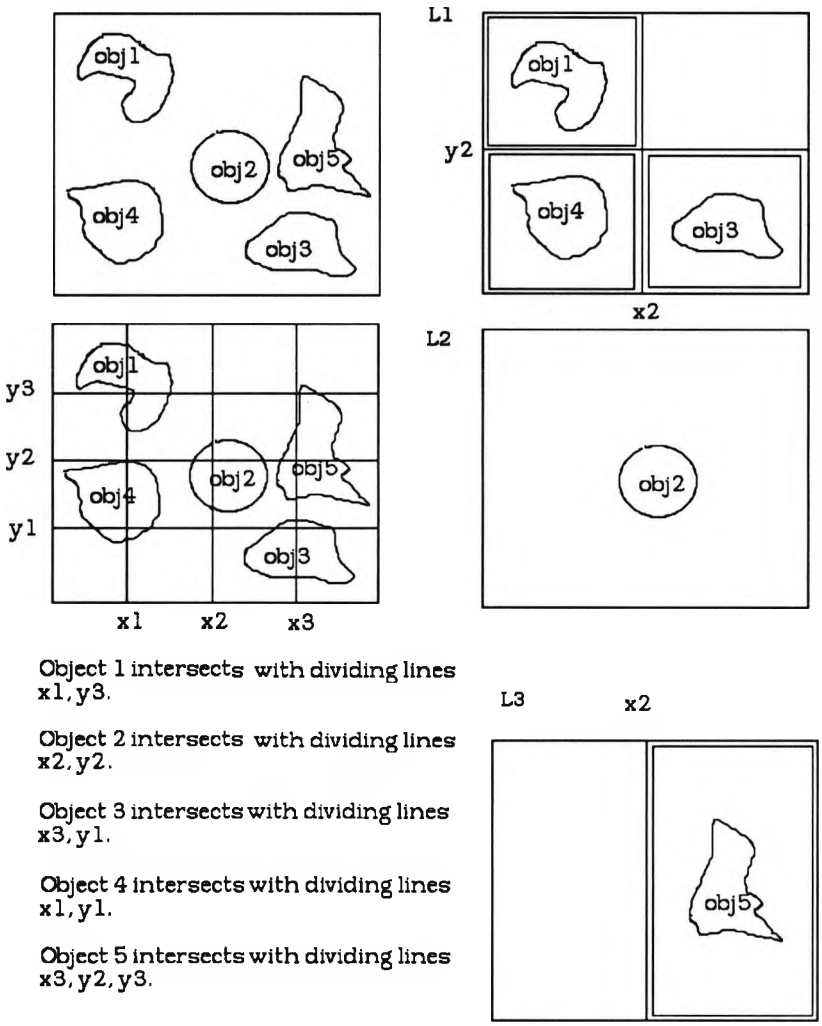


Figure 4.23. A spatial object database in a 2-d space.

(ii) Alterations

Based on the above analysis, a multi-layered grid partition can be introduced. To maintain the storage utilisation the multi-layer grid partition is constructed so that dividing positions on every layer form a complete partition as if there is only one layer.

Let $L^{(j)}$ be the set of dividing lines (planes or super-planes if $m > 2$) for layer j for $j = 1, 2, ..., x$ then the following relation is always true:

$L^{(i)} \cap L^{(j)} = \phi$ for all i and j, where $i \neq j$.

An object identifier calculation is briefly illustrated in Appendix A5 (Figure 4.). Unlike the point data (zero-sized object) space, storing the object depends on the chosen resolution. When a cube size (the minimal rectangle that includes the object) is determined, the number of buckets required can be derived. If the size of an object or the number of buckets used for storing the object is recorded in the index then the total number of the grid cells in the previous layers can be calculated. The result can then be used to adjust the calculation of the object identifier for the current layer. This adjustment is required because an object can occupy more than one data block. It is treated as a VLR data item as objects of different shapes and volumes require different descriptions or representations. The alternative way of dealing with such cases is to multiply the number of buckets required for an object as a weight function to the total number of grid cells. Adding a new object is briefly shown in Figure 4.25.

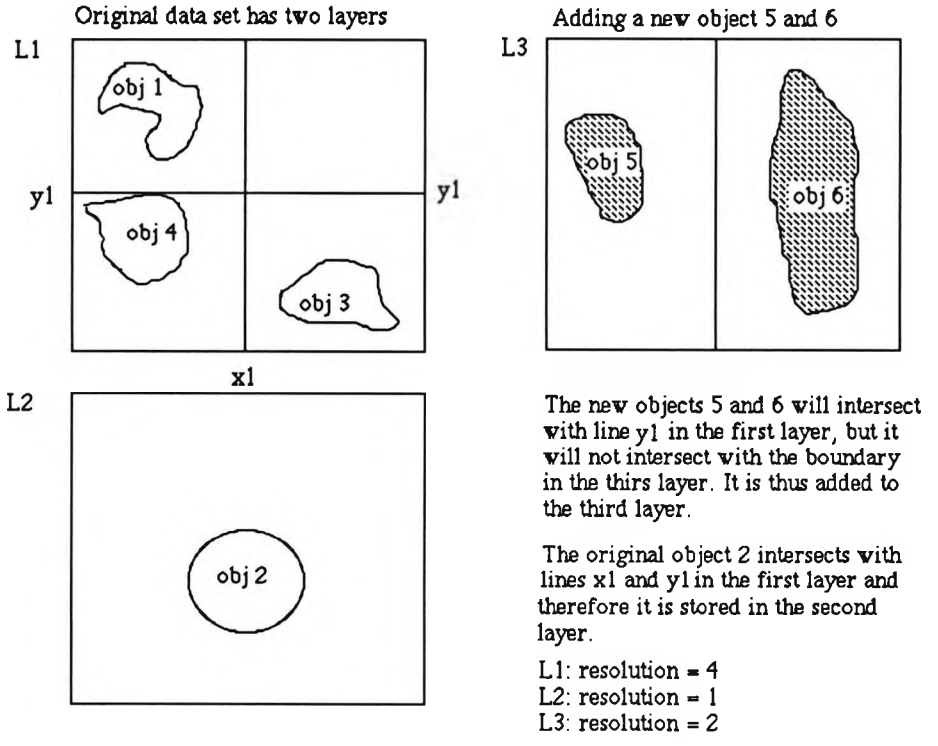


Figure 4.25. Multi-layered grid cells: Adding new objects.

The split heuristics are important to the performance of the algorithm and some rules have to be introduced to avoid ambiguity.

(iii) Multi-layered grid partition implementation considerations

To implement a multi-layered grid partition, a rule as to which layer an object should belong to, needs to be determined. Suppose there are k layers for an object data set and they are labelled from highest layer to the lowest one in sequence as (0, 1, ..., k - 1). If the boundary of the object to be determined is { (ai1, ai2) for i = 1, 2, ..., m } then the steps to follow are:

(a) choose the layer labelled by j starting from 0;

(b) if the object does not overlap with these partition boundaries of layer j then choose j as the home for the object, search or store the object at layer j and do (e); otherwise

(c) choose the layer labelled with (j + 1) and do (b) until all existing layers have been exhausted, i.e. current layer is: j = k - 1;

(d) add a new layer k to the data set to store this object and increase the number of layers;

(e) finish the process.

The same object can sometimes be fitted into different layers at the same time, i.e. it will not overlap with the partition boundaries of other layers. To enhance the overall performance, as many objects as possible should be presented at the highest layer for a given search.

(5) A brief comparison between the BANG-file and the grid file

The major difference between the BANG-file and the grid-file algorithms lies in the partitioning approach. The BANG-file applies m-d cube partitioning to a search space and allows embedded regions in a partition, whilst the grid-file employs (m-1)-d superplane partitioning for a search space. The BANG-file algorithm also presents high storage utilisation for uneven data distributions. The beauty of the BANG-file algorithm is that the index grows with the rate of the data and in allowing the embedded regions in the BANG-file algorithm, it supports localised reorganisation. This locality shows high ability of dealing with dynamic situations. The grid-file displays a predictable performance in terms of access speed. It guarantees a two-disk access in the worst cases because of the one-to-one relationship between an index entry and a grid cell. However, this relationship can be maintained at the price of very large index file for an uneven distributed search space, i.e.

the index file grows at the rates of partitioning.

4.2.6. R-tree and R+-tree Algorithm: ALT[6] = {PT5, SA1}

R-tree is designed for representing spatial objects - non-zero sized objects. The m-d data space is divided into a number of m-d rectangles. The salient feature of the R-tree is that the boundaries of a cell (a region which includes an object) do not need to be predetermined. This feature distinguished R-tree scheme from the above mentioned grid partition methods. At the leaf level of the index, the atomic object is represented by its minimal closure boundaries and the location where the object is stored. At the higher level of the index, each node will contain m rectangles to be covered by this level. Like the B-tree, the higher level covers the range of the lower levels, i.e. the higher level nodes cover a larger rectangle in the data space. Each node in the R-tree is represented by a pair (I, p). Here I is an identifier, which indicates the range covered by the rectangle and p is the pointer to either a bucket number which stores the object at the leaf level or a pointer to the next level in the index at the non-leaf level. As the data space is divided by m-d rectangles represented by its boundaries the domain of a region in R-tree is flexible, in comparison with the Bang file. R-tree does not limit itself to fixed size boundaries. However, this flexibility is gained by storing more information about a region in the implementation, i.e. extra storage space is required. The reason is that the representation of a rectangle indicated in [GU84] uses RECT = {xlow, xhigh; ylow, yhigh} for a 2-d region. For m > 2 representing a rectangle requires even more information. Each dimension needs to specify high and low boundaries, as a result, the R-tree method needs information extracted from (2 x m) values to identify a rectangle.

The implementation may raise problems in that, at the higher level of an index, two different nodes may include the same object at the lower level, overlapping some regions. Consequently, one of them may be searched in a false drop. This indicates that if there are n overlapping regions then the possibility of a false drop during a search is (n - 1)/n = 1 - 1/n, and therefore, the performance deteriorates. R+-tree has tried to reduce the overlapping by introducing more rectangles covering a smaller region, but it ends up with a higher tree index than that of R-tree. Figure 4.26. briefly describes R-tree algorithm.

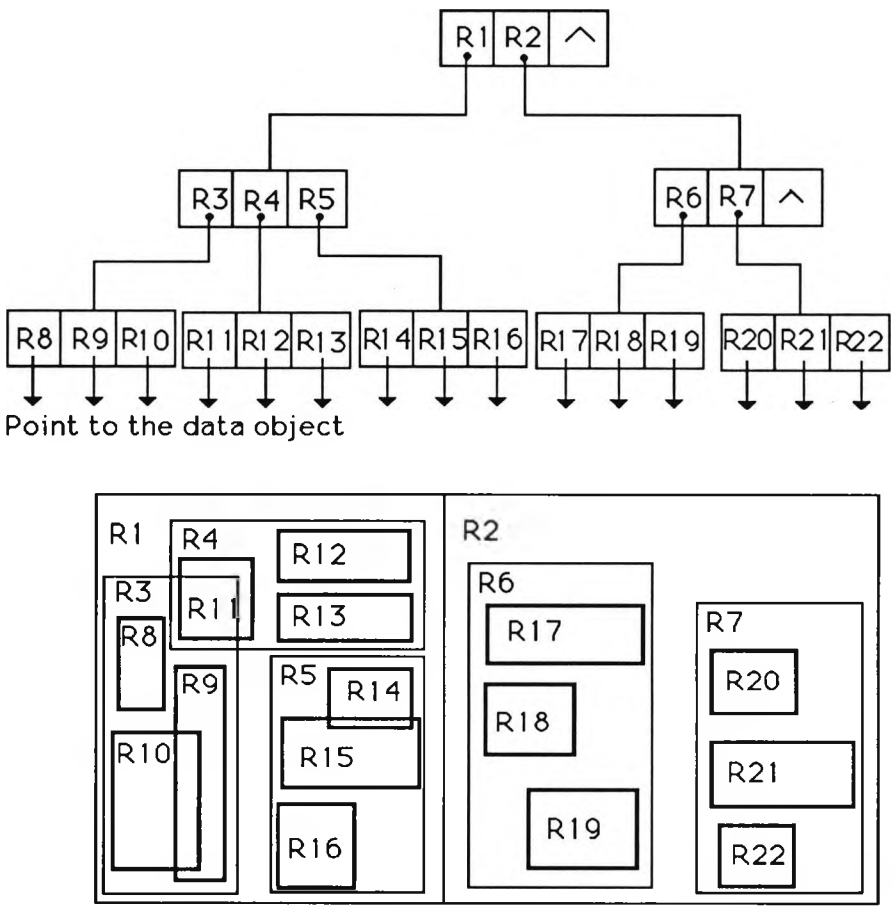


Figure 4.26. The R-tree algorithm.

ALT[6] is suitable for representing a spatial object where the overlapping can be avoided by the nature of the data distribution pattern.

(1) The characteristics which encourage choice of R^+-Tree algorithm

(a) The database concerned is a spatial database : C61.

(b) It is difficult to represent these spatial objects by a standard grid cell : C62.

(c) A large proportion of these spatial objects occupy more than one pages/buckets : C63.

Rule set 6 (Rset6)

(R6.1) if (C61 and C62) then ALT[6]

(R6.2) if C63 then ALT[6]

(2) The reasons for setting these characteristics to choose the R^+-Tree algorithm

The ALT[6] provides a possible solution to various shapes of spatial object representation among chosen algorithms.

(3) The strengths and weaknesses of the R-Tree

(a) The R-tree caters for the spatial objects with varied shapes and positions.

(b) It provides an alternative to an object-oriented database.

(c) It adopts a B-tree as its access paths implementation so that it copes with dynamic situations efficiently.

(d) It represents identification of an object by its boundaries explicitly so that when $m > 2$ the object identification may occupy a large amount of storage and consequently, increase the index size.

4.3. The Application Abstract Profiles (AAPs)

The AAPs are abstract types of applications described by their properties. These properties are used to classify the types of applications in order to match them to a suitable implementation algorithm. They are built up by the expert system based on previous database applications. New knowledge of later applications can be added to the system when required. The knowledge is stored and used to find solutions to a new application based on the similarity comparison between features such as the

main access mode, the main attributes used in the queries, and the data distributions. These features can have different emphasis for various applications. Since the knowledge of applications is complicated and plays an important role in choosing algorithms we will define these AAPs and describe how we can use them to tune and monitor the performance of a database.

As explained in previous sections, the premises in the rule base are application-oriented. We use the AAPs to represent knowledge about applications. To serve the purpose of tuning physical database design these profiles are utilised to classify a given application. These AAPs are thus designed for comparison with a real application. They will also be used as input data to performance evaluation to refine algorithm selection and adjust boundaries for various features which are employed to classify applications. These profiles are described by a set of abstract properties from the real data set. The collection will be expanded as new application profiles are added to it. After performance evaluation the results will be stored as part of the knowledge for these application profiles so that they can be reused after an approximate correspondence (similarity) has been identified between an application and an AAP.

The AAPs consist of a set of application classes AAP = {AC1, AC2, ..., ACx}. Each class is described by a set of properties. These properties have four aspects: application types (AT), data distributions (AD), query types (AQ) and performance evaluation (PE). They vary for different types of applications, especially data distributions and query types, and change during the life-time of a database. To form these properties major factors reflecting data distribution and queries have to be carefully chosen. An application **type** is defined as:
ACi = { ATi, ADi, AQi, PEi } for i = 1, 2, ..., x, x is the number of data sets in an application.

ACi Application class
This defines application class, which includes an application domain and tasks to be performed.
ACi = { application area, definition }

Application area is identified by the nature of applications. 'Nature' means applications with the same underlying structures of a search space and query

patterns. The dimensionality of a search space is determined by query frequencies and this depends on the experience of functionality for an application system. A search space also relates to the data pattern. An expert system can gather such experience and the data patterns. An example would be transportation applications. The transportation means can be different - by sea, by train, or by air. It does not make much difference for applications concerning scheduling, booking and cancelling seats and so forth. The definition gives data sets and functions involved in an application. An application class can be a library application with data sets about the book, the borrower and the supplier; with functions about lending, returning, querying, adding, deleting, and searching for a book. An ACi can also be a selling business application with data sets on stock, customer, staff, and supplier; with functions of distributing, purchasing an item, and order processing. It can also be an airline application with data sets for flight, traveller; with functions of booking, cancelling of seats and scheduling flights, etc. Each application class represents a set of 'similar' applications with either generic features or functions. For instance, library A and B are the same application class with similar application characteristics ignoring whether they are video libraries; music libraries or book libraries or a mixture. Similarly, for the selling business and airline application the system will assume that selling companies A and B or airlines A and B belong to the same application classes, regardless of what goods the businesses sell or the destinations an airline serves. Once an application which represents an application class has been run on the system all necessary information in the form of an AAP will be recorded for the system to use. We will see later that an AAP can be used to derive data from incomplete information supplied, to guide the determination of dimensionality of a search space, to be a data resource for analysing salient features of applications, and to draw a similarity comparison.

AT Application Type

AT = {ATi for i = 1, 2, ..., x}

AD A set of data distribution for an AAP

AD = { ADi for i = 1, 2, ..., y }

ADi = (m, r, b, Ddis)

where:

m - dimensionality (we may assume m = 2).

r - resolution measured by the number of grid cells in an equal-

sized grid cell partition.

b - relative bucket size is the bucket size that will satisfy a chosen resolution.

Ddis - data distribution, relating to C = {C[i] for i = 1, 2, ..., r } from which a group of parameters can be derived and some of them stored for system usage. These parameters will be described after a discussion of algorithm AD.

AQ A set of query type for an AAP

AQ = { AQi for i = 1, 2, ..., z }

AQi = (Qi, F, Ps, Rs)

where:

- Qi - query type measured by query frequencies.
- F - properties of attribute set by a weight function f: F = f(A1, A2, ..., Am) = {fi for i = 1, 2, ..., m}
- Ps - point search rate
- Rs - range search rate

The function F is evaluated by the percentage of a particular attribute occurrences in queries and the total number of queries made.

Dyn = {Ir, Dr }Dynamic Features

where:

- Ir - insertion rate
- Dr - deletion rate

PEi = (T, Su, ATL[x])Performance evaluation

where:

- T - average time required for retrieving a data item.
- Su - storage utilisation.
- ATL[x] - an algorithm chosen for implementing the grid partition.

In the application abstract profiles m is relatively static whereas data set size n, data distributions Ddis and query properties Q have dynamic features; the bucket size b and partition resolution r are either set by a user or limited by the operating system as software constraints. Among these parameters, n and b indicate the size of the physical database. As will be mentioned below, the data pattern is size

independent. Thus n and b are transformed to a system derived scale to form a standard scale for comparison. Different data distribution and query patterns are developed from applications to classify a 'new' application. Here 'new' has a double meaning: firstly, an application with the same class may be stored in the system as an AAP so that the application appears to be new (although it is not new in the sense of an application class); secondly that the application type is not in the AAP so that relevant data have to be extracted to represent a new application class in the system. In the first situation, only brief information of the new application needs to be stored, such as data set size, the application name and dynamic features. Information about data distribution and query are inherited from the previous instance under the same class.

Data distribution Ddis reflects the pattern of data items in a given data space. A data space (region) can be divided into a number of data subspaces (subregions). If one data subspace contains more data items than the other then we say that the data is more densely populated in the former subregion than in the latter. The density can be measured by the number of data items in a standard partition of the data space. Now we define the partition and how we approximately measure the data distribution.

A partition which is used for distribution measurement is defined as an equal-sized grid cell partition over the data space. To facilitate the measurement we map the concerned space into a 1-d space by applying the z-code [OR86] to represent each grid cell in the data space. Based on the given parameters of data concerned, the resolution of the data space can be determined. With the given data set the distribution is modelled.

As soon as the resolution r is determined a one-to-one mapping is established between the z-code and the regions introduced by the equal-sized grid cell partition. In order to know the properties of a given set, the data items are used as input data to the profile, and as a result, the data distribution information will be the output. The data distribution will be measured using the following parameters:

Nover - the number of grid cells which have more than b data items;
Nempty - the number of grid cells which have no data items;
Nmax - the maximum number of required data blocks;
Nmin - the minimal number of buckets estimated for storing a data set.

The algorithm of the data distribution profile AD

Research has been done to construct various data distribution profiles [RE84] to study the behaviour of performance of a database. Most of them are at a theoretical level with a chosen distribution profile containing assumptions such as uniform distributions or the binomial or Poisson distributions, or randomly generated data, etc. simulated for research purposes. In an actual database, however, data attributes usually have functional relationship or correlations so that the above assumptions may not represent the nature of the data set in use. The reason is very simple: a data item is a collection of property information about an object in the real world, categorised as something that an organisation is interested in. For instance, data kept for an airline business may be concerned with storing information about distances and prices. These two attributes follow certain distribution - as distance increases the price will also increase. An undergraduate student file tends to have a narrow age scope highly populated as most students are aged between 18 and 21. The levels of employee's position (top managers, middle level managers, and employees) may be strongly correlated to the size and the nature of an organisation. This indicates that data distributions of many cases, in practice, do correlate to the nature of an application, i.e. they are application-oriented. Similarly, the query pattern, in real databases, may be a function of time. A manufacturer storing customer information for order processing may relate the query pattern to seasons and this pattern would be a time function. The query pattern of a database used for financial applications may be influenced by the financial cycle. Data distribution and query patterns vary for different types of application, but they may also be close enough for similar business organisations where the differences can be ignored. Data distribution is usually determined by the nature of information required to carry out functions of an organisation; and query patterns often relate to the activities of an organisation. To allow the system to recognise the data distribution pattern different AAP profiles are stored as criteria for making comparisons. These profiles are constructed by abstracting features of applications and by classifying them thereafter. Comparisons are conducted by heuristics and judgement. The equal-sized grid cell partition is used to analyse and describe the data distribution of a m-d data space.

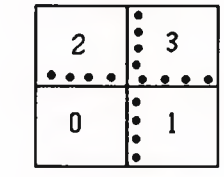
Information about AC

AC is defined as application classifications, which is a set of application classes

ACi for i = 1, 2, ..., x. Information about ACi and other elements in the AAP has to be collected from real applications and refined by the system. Limited by time, this task is beyond the scope of this project. The task involved is to investigate real applications of different kinds, carrying out an analysis which represents data and query patterns, algorithms, and performance gearing towards the AAP definition. ACi abstracts application classes and attaches a definition to each of them.

The algorithm for data distribution AD

In order to estimate the data distribution features of an application the data pattern needs to be captured. By pattern we mean the shape of a data distribution. One of the pattern recognition techniques involves having a set of standard shapes coded in a way which allows us to draw comparisons. Data, in the view of geometrical space, form a pattern. Different distributions can be seen as different patterns which form different shapes. To categorise data distribution we need information which allows us to recognise a shape at a specified resolution level. Here, the pattern is an abstract concept. Its size is irrelevant but it is associated with the resolution. A resolution is scaled in terms of the number of grid cells in a data space, and the space size can differ. When the resolution changes, the pattern we perceive will also change; whereas with the same resolution we perceive the same pattern with a different size as the same shape in each grid cell (a system unit). This phenomenon can be seen from Figure 4.27. For our purposes we use an absolute scale to measure similar patterns with different sizes, i.e. we view the shape as an absolute standard, producing a **fixed** number of grid cells, gearing it to different sizes of applications and the constraints for classification. These standard data patterns are created by generating the AAP from analysis of applications.



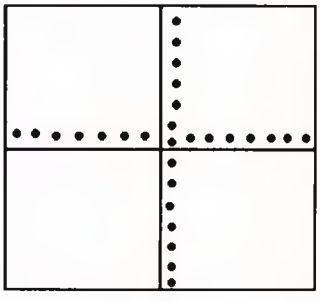
$n = 16$
$Ncell = 4$

$\overline{b} = \frac{n}{Ncell} = 4$

$C[0] = 0$
$C[1] = 4$
$c[2] = 4$
$C[3] = 8$

$Nempty = 1$
$Nover = 1$

$\left\lceil \frac{C[3]}{\overline{b}} \right\rceil - 1 = 1$ (overflow depth)



$n = 28$
$Ncell = 4$

$\overline{b} = \frac{n}{Ncell} = 7$

$C[0] = 0$
$C[1] = 7$
$c[2] = 7$
$C[3] = 14$

$Nempty = 1$
$Nover = 1$

$\left\lceil \frac{C[3]}{\overline{b}} \right\rceil - 1 = 1$ (overflow depth)

They belong to the same type of data distribution. They are measured by overflow depth, relative buckt size $\overline{b}$ and Nempty.
Overflow depth is the extra number of buckets an overflow data bucket required.

$$\text{overflow depth} = \frac{\sum_{i \in over} \left(\left\lceil \frac{C[i]}{\overline{b}} \right\rceil - 1 \right)}{Nover}$$

$i \in over$ refers to these cells which have more than $\overline{b}$ data itemd.

Figure 4.27. Similarity Comparison.

For a given application we should know the dimensionality m, and an estimated n and b. The data set itself may need to be built up dynamically or it is an input at the data entry stage. Firstly an estimate of resolution is produced by n/b so that r can be chosen at the right level to start with. After choosing r we can then calculate the rest of the parameters to determine the application type for its data distribution features.

Here we set standards here for distribution comparison based on the idea of shape recognition. Each type of data distribution is added to the system from an analysis of real data. The AAP acts as a shell which allows classification knowledge to be added to it. The data distribution is measurable as follows.

(1) Resolution r

As defined above r reflects the number of grid cells for a 2-d data space. For an equal sized grid cell partition r = sx * sy, where sx is the number of slices in the x dimension and sy is the number of slices in the y dimension. Initially the system can choose a set of ranges:

r = { 256, 512, 1024, 2048, 4096, 8192, 16384 }

i.e.

(sx, sy) = { (16,16), (16, 32), (32, 16), (32, 32),
(64, 32), (32, 64), (64, 64), (128, 64),
(64, 128), (128, 128) };

The resolution r can be chosen with different values for the same data set to reflect different levels of resolutions and only information about the highest level resolution (at the highest resolution value) needs to be stored. For other levels of resolutions required data can be easily derived from the highest level of r by summation. In practice, the value of resolution is function of bucket size b. i.e. similarity comparison has to be considered at the resolution level of bucket size b to be meaningful.

(2) Bucket size

b = { 512, 1024, 2096 } (bytes).

(3) Number of grid cells which have more than b data items against the total number of grid cells. (overflows)

$$\frac{\text{Nover}}{r}$$

(4) Number of grid cells which have no data item against the total number of grid cells. (data holes)

$$\frac{\text{Nempty}}{r}$$

(5) Storage utilisation estimation

$$Su = \frac{Nmin}{Nmax}$$

To compare an application we first estimate resolution level and choose the nearest one set by the system.

Given n, b we can estimate

$$r^{(e)} = \frac{n}{b}$$

Compared with r we will choose the resolution satisfying the following condition:
$r = \{ x \text{ where min } \{(x - r^{e}) \text{ for all } x \in r \} \}$
then we represent the real data space in the partition of a chosen r. Given m = 2, D = Dx X Dy, where Dx = Dxmax - Dxmin, Dy = Dymax - Dymin and r, a one-to-one mapping can be established between identification of a grid cell in the partition and the data space D. The mapping is carried out by the following calculations based on a given data set Ds: suppose Ds = {d1, d2, ..., dn} where di = (dxi, dyi). The length of the intervals in the x dimension will be:

$$Ix = \left\lceil \frac{Dx}{Sx} \right\rceil$$

Similarly, the length of the intervals in the y dimension will be

$$Iy = \left\lceil \frac{Dy}{Sy} \right\rceil$$

Given a data item (dxi,dyi) its corresponding grid cell (xi, yi) can be determined as:

$$xi = \left\lfloor \frac{dxi - dxmin}{Ix} \right\rfloor = x0x1x2x3x4x5x6x7x8,$$

$$yi = \left\lfloor \frac{dyi - dymin}{Iy} \right\rfloor = y0y1y2y3y4y5y6y7y8$$

Subsequently, its corresponding z-code can be derived as:
z(xi, yi) = x0y0x1y1x2y2...x8y8 or
z(xi, yi) = y0x0y1x1y2x2...y8x8.

Depending on the chosen resolution r the maximum length of the z-code (|z|) is $\lfloor \log_2 r \rfloor$. We will need r and the value in counters C[i] for i = 1, 2, ..., r to capture information about the data distribution. In order to transform the real data space we use the same resolution. The absolute space is the one which interprets each grid cell as a unit. Having decided the resolution the following algorithm can be used to estimate the data distribution.

Algorithm

INPUT

Data set:	Ds = {di |di = (xi,yi) i = 1, 2, ..., n}
Bucket size:	b
Data space size:	Dx, Dy
	Dx = Dxmax - Dxmin
	Dy = Dymax - Dymin
Data space resolution:	sx, sy (number of slices in x and y dimension for the partition)

OUTPUT

BRIEF INFORMATION

Nover	- number of overflow cells
Nempty	- number of empty cells
Nd	- overflow depth
Su	- storage utilisation
d(even)	- data distribution
Nmin	- minimal storage
Nmax	- maximum storage

$\frac{Nmax}{b}$ reflects the data distribution. If $\frac{Nmax}{b}$ is larger than r the data are less evenly distributed. These parameters which are derived by approximate calculations such as Ld(xi), Ld(yj) for i = 0, 1, ..., $\lfloor \log_2 r \rfloor - 1$, j = 0, 1, ..., $\lfloor \log_2 r \rfloor - 1$, Nempty, and Nover from C[i]. They are calculated in the following algorithm.

DETAILED INFORMATION

The number of data items in each grid cell: C[1], ..., C[r].

ALGORITHM (Pseudo code)

Suppose the number of grid cells created by the partition is r, and C[1], C[2], ..., C[r] are counters used to calculate the number of data items in each grid cell.

```
AD = {
        while (input data set is not empty)
        {
        read a data item from an input data set (x, y);
```

$$X = \left\lfloor \frac{x - xmin}{Ix} \right\rfloor;$$

* calculating the slice number in the x dimension *

$$Y = \left\lfloor \frac{y - ymin}{Iy} \right\rfloor;$$

* calculating the slice number in the y dimension *
i = interleaving X and Y for their binary representation;
C[i] = C[i] + 1;

}
* initialisation *
Nover = 0;
Nempty = 0;
Nd = 0;
d(even) = 0;
for (i = 1; i < r; i++)
{
    if (C[i] > b)
    {
        Nover = Nover + 1;
        d(even) = d(even) + C[i];
        if (C[i] > Nd) then Nd = C[i];
    }
    if (C[i] == 0)
    {
        Nempty = Nempty +1;
    }
}

Nmax = Sx x Sy + Ntot x min(Sx, Sy);
Nmin = n / b + 1;
d(even) = (2 x d(even)) / (Nover x b);

Nd = Nd/b;
Su = Nmin / Nmax;
store Nover, Nempty, Nd, Su, d(even) Nmin and Nmax to DBPi for the data set;
}

For a given data set identified by its data set name, the detailed information of the profile can be stored as a separate layer of knowledge about an application. It reflects approximately how the data items are distributed in the data space. The value in each counter will be retrieved if the detailed information of the application profile is required. The set of values can also be used to draw the distribution pattern diagram ordered by the z-code as the horizontal axis and the value of each counter as the vertical axis. A sixteen grid cell diagram is shown in Figure 4.28. The pattern can be used to help experts or database designers to recognise different data distribution better so as to aid building new knowledge such as a new class of application into the system.

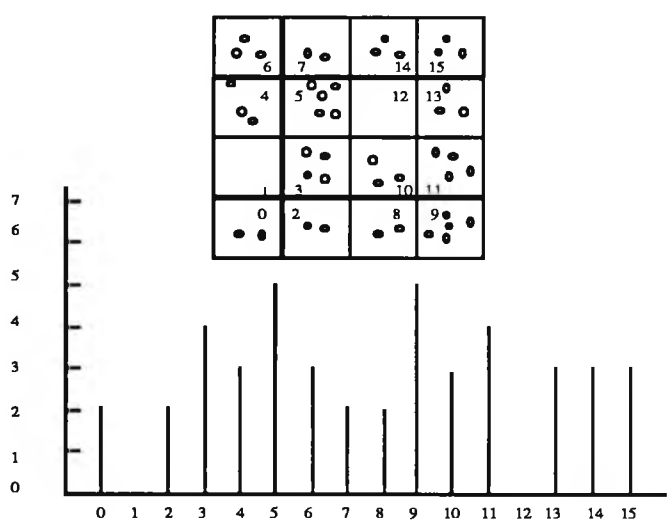


Figure 4.28. data distribution pattern.

The algorithm of the query type AQ

INPUT

The number of attributes or the dimensions of the data space: m.
Each occurrence of an attribute: Nj (j = 1, 2, ..., m).

OUTPUT

Frequencies of each attribute: Q[i] for i = 1, 2, ..., m.
Point and range search rates: Ps and Qs.

ALGORITHM

Assume that the DBMS can provide the number of times an attribute is requested for each operation (an operation can be a group of queries).

```
AQ = {
        if the database status is new then initialise the counter used for the
        number of frequencies of each attribute: Q[1], Q[2], ..., Q[m] and
        number of point search Pc, number of range search Qc, number of all
        search Ac;
        calculate the query frequencies by accumulating the times of occurrences
        for each corresponding attribute;
        if the query of an operation includes a search for attribute i Nj times then
        increase Q[i] by Nj;
        calculate the point and range search rate by accumulting each type of
        seach Pc, Rc and all searches Ac;
        for every search
        {
            Ac = Ac + 1;
            if a search is a point search then  Pc = Pc + 1;
            otherwise  Rc = Rc + 1);
            Rs = Rc /Ac;
        }
    }
```

The knowledge about the query frequencies provides data for query frequency controlled splitting. The frequencies can be described by weighting values. The weight can be calculated by the formula: fi = Wi / Wmin for i = 1, 2, ..., m. Where Wi is the frequency for attribute i recorded by the system and Wmin = min { W1, W2, ..., Wm } = min (Q[i] for i = 1, 2, ..., m). If the performance deterioration is

recognised by the system then the data about query frequencies can be referenced and used to choose a split strategy when required.

The algorithm for frequency weighting calculation

Suppose Q[1], Q[2], ..., Q[m] are the number of queries for attributes 1, 2, ..., m involved in the search respectively, the algorithm for weighting estimation is illustrated below

```
estimate-q-weight()
 {
      sort Q = { Q[1], Q[2], ...,  Q[m] } to be
      Q' = { Q'[1]', Q'[2], ... Q'[m] | Q'[i] < Q'[j] if i < j };
      * In set Q, the element Q'[1] = Wmin and we assume it is not equal to 0
      otherwise the attribute will be ruled out from the search space since it is
      irrelevant to queries. *
      f[i] = Q'[i] / Q'[1] for i = 1, 2, ..., m;
      * if max { f[1], f[2], ..., f[m] } = f[x]} then the attribute x is the dominant
      attribute, i.e. the distribution chosen to be split by a frequency controlled
      function. *
      store f[i] for i = 1, 2, ..., m;
 }
```

the algorithm of the performance evaluation for an application

INPUT

Data set: Ds = {d1, d2, ..., dn |di = (xi, yi) i = 1, 2, ..., n}.
Implementation algorithm: ALT(id).
Here id identifies an implementation algorithm.

OUTPUT

Average search time: T
Storage utilisation: Su

ALGORITHM

```
PE = {
      partition the data set based on chosen implementation algorithm;
      calculate r = n / b;
      calculate Su = r / B;
```

```
        * where B is the number of buckets actually used *
        generate a data item within the data space to be retrieved:
        ( d = { a1, a2, ..., am });
        record the point search time t1 by benchmarking;
        generate a range search query:
        (r = {(k11, k12), ..., (km1, km2) ∈ D }
        calculate the accuracy ac of the range search;
        ac = no. of data items requested / no. of data items searched which satisfy the
        query;
        * calculate the number of buckets need to be covered by the partition for the
        range search *
        store the total time used for the range search;
        store the total number of data items requested;
        * calculate the average range search time for a data item *
        t2 = total time used / total items searched
        * calculate the average search time *
        T = (t1 + t2) / 2;
        * store the results in the database profile *
        store point search time: t1;
        store range search time: t2;
        store average time: T;
        store range search accuracy: ac;
        * ac reflects the quality of partition for a range search *
        store storage utilisation: Su;
        * for different ALT[id] the calculation will be different *
}
```

The performance is evaluated for an application which has already been implemented by a chosen algorithm. The result is compared with a chosen AAP and the outcome of the comparison is used for adjusting rules. For different implementation algorithms the performance evaluations are detailed in Appendix A6.

The similarity comparison

To capture the features of a data set, the process is analogous to a m-d matrix. The data distribution can be reflected by the zero and non-zero elements if each element indicates the number of data items and the position in the matrix indicates the position

in the given space. For an even distribution the condition is such that the value of each element in the matrix approximates to the size of buckets b or the number of elements which are over the value b and the number of zero elements are small. Based on this idea the degree of an even data distribution can be measured by the accumulated differences between the number of data items in a grid cell and the bucket size. Similarly, to represent a sparse matrix and a full matrix in the computer different methods have to be used in order to store a full matrix or a sparse matrix efficiently. To store data with different data distributions different partition strategies are considered to cater for different requirements. If we use the example shown in Figure 4.29. we can store the graphic diagram for a data distribution in an array:

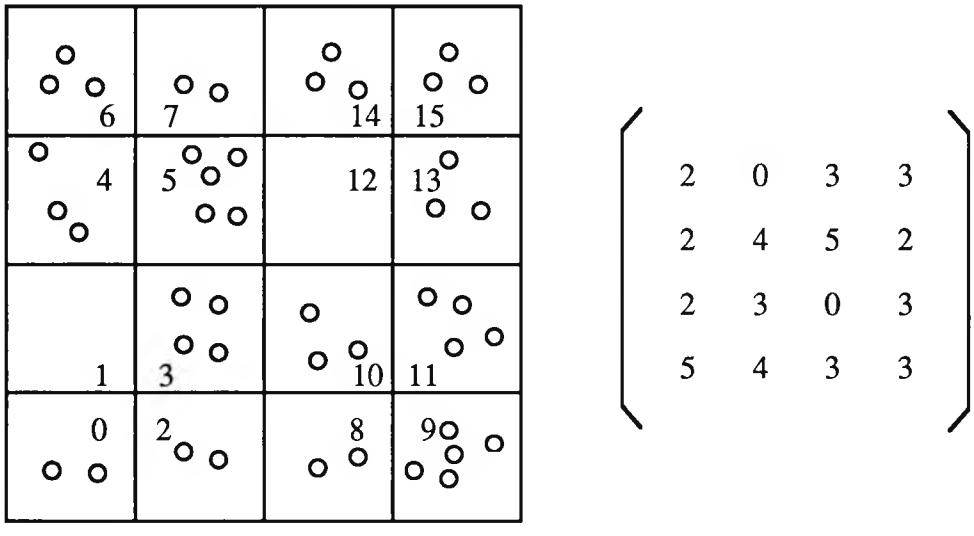


Figure 4.29. Array representaion for a 2-d search space.

To represent data distribution in such a way, matrix transformations can be applied to symmetric pattern recognition. The reason is that symmetric patterns can be regarded as having the same data distribution features.

The degree of similarity is defined in the rule base. The function of the similarity

comparison is to derive the deviations between the data about the database and a chosen AAP and then inspect the rule base to identify the degree of similarity.

ALGORITHM:
The similarity comparison algorithm is invoked by a system trigger when an unsatisfactory performance is detected.

```
similarity()
{
        search the database profile for the data set concerned;
        construct a profile for the application in the form of the AAP;
        search the AAP for an application type;
        calculate the deviation of the chosen AAP from the application profile;
        search the rule base to decide the degree of the similarity;
        search the rule base for the recommended implementation algorithm if satisfied
        similarity is found;
}
```

The algorithm is a narrative which will be detailed by the rule base.

4.4. The Rule Base

The rule base describes how the information stored about a database is used. It gives criteria to classify application types and provides means to choose a particular implementation algorithm or split strategy. Classification of an application is the basis for selecting an algorithm. The algorithm selection is similar to a medical diagnosis. The expert system needs to classify applications based on their features according to the stored profiles, reflecting past circumstances, in order to choose an algorithm of physical organisation. Similarly, a doctor needs to classify his patients based on their actual symptoms according to his stored knowledge, which reflects his experience and expertise developed in the previous cases, to give a prescription. For each application the system stores a set of data which indicates its characteristics. Those applications with similar features to a stored AAP are recognised by the system as the same application type. A new application can thus be identified by the system decision rules, classified as an existing application type, and processed accordingly.

4.4.0. Matching revisited

One of the heuristic problem solving techniques is heuristic matching. This matching process explores a solution by identifying similarities between features of a known problem (experience, learned domain) and that of a new problem (to be tackled). This approach is useful especially in situations where the features of the new problem cannot be determined when a solution is needed; or by analysing pros and cons of a particular solution against the features of the new problem, placing emphasis on salient features of the problem and tackling it by knowledge of a particular solution which is designed to cater for these features. There are a variety of matching techniques devised for pattern recognition and image processing problems [HE88] [HO88] [PL84] [DE78] [SH78] [FO88] [WO78]. However, there are only a few schemes [WA84] [MI90] employed for problem solving. In this section, we review various matching techniques briefly and concentrate on how heuristic matching can be deployed for problem solving.

Several commonly used approaches to knowledge representation are: the logic representation [FR86] [JA89], production systems [VA89] [FR86], frames [VA89] [FR86], object-oriented representation [BU89] and semantic networks [FR86]. Each representation approach is suitable for a range of problems. However, each approach involves the concept of matching, especially frame and object-oriented

representations.

The concept of matching is based on the idea of discover similarities /distortions among two or more structures /objects /subjects /relations/ patterns, etc. The matching techniques are mainly used in image or pattern recognition and production interpreters [HE88] [FO82] [FO88] [HO88] [SH87a]. For pattern recognition the problems to be solved are to find similarities between a set of shapes (known pattern) stored in a knowledge database and an object to be recognised (new pattern). These stored shapes are represented by their features and used as goal status or samples; and these objects are compared with these samples/shapes for a match. A match can be exact/clear and inexact/fuzzy. Here the word 'exact' depends on a set of goals. For example, match can be done on size, knowledge structures, topological structures, features, etc. Some patterns can be described accurately by mathematical formula, such as a rectangle, a circle, a polynomial curve, a sine curve and so forth. However, the difference between a mathematical representation and a pattern match is that the former cannot cope with distortions and irregular shapes efficiently. This is because irregular shapes are difficult to represent accurately, distortion to a regular shape cause irregularity. The latter does better.

Few matching techniques have been employed extensively to A.I. problem solving [FO82] [WA84] [MI90]. However, there are demands to develop such techniques in the domain of certain range of problems, where an efficient solution can be introduced by heuristic matching. For example, there are situations when an expert system is used to find a match between a set of known environment profiles and a unknown profile, where the match conditions are based on experience. In addition, a unknown profile is dynamic. In such situations, heuristic matching techniques can be effectively applied to implement the match process. The reason is that the complexity and dynamic features can be handled by introducing matching rules, which are based on heuristics. Heuristics avoid and eliminate extensive details and explore optimal paths to a solution. A real-time fault diagnosis system can involve combinations of various possible faults that are too complicated. In such a system, heuristic matching can be introduced to implement this diagnostic control. This is done by accumulating fault knowledge acquired previously and representing the knowledge as a known profile or by analysing critical fault conditions and storing this knowledge as a set of profiles in the knowledge-base. These profiles are deployed to match with a real-time collected sample data to predict if a similar fault

is going to occur. Heuristic matching is carried out by analysing a profile (a profile = features + relationships) of a problem domain and comparing this profile with a set of known profiles (either from real world or from results of simulations) in the knowledge-base to find one which satisfies a set of heuristic conditions for a match. This analysis distinguishes the salient features from other less significant features and treats them separately, either by attaching different weights to them or by searching structure construction, differentiating major factors from minor ones.

In this section we mainly review match techniques and discuss how matching can be applied to inference processes.

A brief literature survey

There are various matching strategies [FO82] [GO77] [MO84] [DA88a] [DA88b] [HO88] [MI86] [PL84] developed in the last decade. A particular strategy is suitable for a set of problem domains. Some problem-solving processes concentrate on the induction process. Some focus on the reasoning process. To perform problem-solving process as a variety of representations can be introduced depending on more or less the strategy to tackle the problem of interest. These are transformation - from initial state to a goal state; matching - mapping problem features to a solution space, etc. For example, analogical reasoning aims at the transformation from one status to another one in interest; approximate reasoning emphasises (1) inversion, (2) aggregation, and (3) cascading; and heuristic reasoning emphasises on experience, and educated guesses. In this section, we place emphasis on matching strategies.

Matching is mainly classified into several categories: template matching [GO84] [HO88] [MO84] [DA88a] [DA88b], where the matching process is based on a predetermined profile - the representative/template frames; partial and best matching [WA84], where descriptions of two or more objects are compared; a feature extraction matching [LE88], where an object to be matched is decomposed into several sub-objects or primitives and extracted features of each sub-object are deployed for this matching process; probability matching is where probability function is introduced to implement the matching process, predicting the possibility of a pattern likely to satisfy a match condition; RETE matching [FO82], applying a OPEN and a CLOSE set to match goal by removing an element from OPEN set to CLOSE set through evaluation of working memory of a production system to

update conflict set; and TREAT matching, which is an improved variant of the RETE matching, removing redundancy introduced by RETE matching via a hierarchical layers based on operation types (addition and deletion). Heuristic matching can be a new candidate to this classification, which utilises the rule of thumb to implement the matching process, in contrast to the exact matching algorithms, to tackle problems with dynamic, complex, unpredictable features and allowable error rates (inexact/fuzzy match).

In recent research [BU89] on expert systems, matching mechanisms have been discussed in two contexts: first in a variety of research on pattern recognition and secondly in research on inference strategies. The author states that:
"Mechanism with pattern matching is a basic mechanism. In rule-based systems the inference engine uses it to apply a rule. In an object centred representation, matching is to be able to compute all the objects which are instances of a special class called filter. In mixed systems, matching is used to find objects on which the inference engine must apply a rule."

Pattern matching and image processing

Pattern matching techniques are mainly applied to pattern recognition. But the analogical reasoning problem solving can also use these techniques. Visualising a problem solving process, a solution can be based on: (1) an analysis of a known problem features, namely, a new pattern; (2) the objective for seeking a solution, i.e. allowable distortion; and (3) a satisfactory solution for a similar situation, that is, an existing pattern, which forms the basis of recognising the new pattern.

Template matching

Template matching [GO84], developed for image processing, compares a targeted frame to a group of predetermined templates and calculates accumulated errors. A threshold is given to measure the acceptability of accumulated errors. Template matching is a calculation-intensive process. It can only be used in predictable situations where the likely image/pattern to be identified is known beforehand. It cannot handle unpredictable dynamic situations because the templates / windows, which are used as criteria to identify an unknown image, have to be stored to do so.

Partial and best matches

Partial and best matches and their application fields have been described and identified in [WA78] respectively. Partial and best matches are defined in [WA78] (page557) as " a comparison of two or more descriptions that identifies their similarities. Determining which of several descriptions is most similar to one description of interest is called the best-match problem". Furthermore, the author has stated that "Partial and best matches underlie several knowledge system functions, including analogical reasoning, inductive inference, predicate discovery, pattern-directed inference, semantic interpretation, and speech and image understanding". However, the author also pointed out that partial-matching is both combinatorial and ill-structured, admissible algorithms are elusive. Tree-matching [CH84] can be classified into partial and best matching category. Tree-matching represents the objects to be compared by tree structures. The matching is carried out by transforming one tree to the other. The similarity is measured by distance. A distance between two trees is defined as the minimum number of basic operations necessary to transform one tree to the other.

Example

Given two object patterns A and B, a partial matching is an A into B matching. A best partial matching is the matching which assigns the highest number of objects in A in to B. i.e. a maximum number of objects has been matched from A into B.

RETE matching

RETE matching [FO82] is developed for production system interpreters. It is claimed that it is an efficient algorithm to find all objects which match each pattern. In a production system, RETE matches the content of the working memory with the rule set to change the conflict set. It represents the rule set as a tree structure and the LHS is SELECTED and JOINED as resulting set to update the conflict set. An example can be seen in [MI90] pages20-24.

TREAT matching

TREAT matching is a variant of RETE with performance improvement in terms of memory and search speed. TREAT aims at reducing redundancy of memory. Different levels of status in the working memory are categorised for old partition, add partition, delete partition. Delete is dealt with by matching the deleted element

with the table in the conflict set. The table which contains the matched element is then removed. The algorithm is described in [MI90] page 30 and examples showing TREAT in action can be referenced from the same book on page 29-34.

From the above review we can see that matching, in essence, is to find EQUALITY among the concerned objects. This equality can be either measured by similarities or dissimilarity. Matching efficiency depends on a well-structured representation. For example, tree representation is suitable for waveform image [CH84] because the waveform has peaks corresponding to nodes of tree structure, but may not be efficient for others. A multiple template matching, which employs a tree structure [LI86] outperforms sequential matching in terms of speed. A transformation concept is easy to implement for small-scaled matching because a small matrix representing a structure could be easily stored in main memory for comparison, but it may be not that easy for a large-scaled matching.

An example:

Problem

Matching two small objects to find out distortion.

Representation

Two dimensional matrix for A and B respectively:

$$A = \{a_{ij} \mid a_{ij} = 1 \text{ if A has } j^{th} \text{ value for attribute } i;\ a_{ij} = 0 \text{ otherwise}\}$$

$$B = \{b_{ij} \mid b_{ij} = 1 \text{ if B has } j^{th} \text{ value for attribute } i;\ b_{ij} = 0 \text{ otherwise}\}$$

for i = 1, 2, ..., n; j = 1, 2, ..., m. Where n is the number of attributes, m is the maximum number of values the attribute i assigned.

To match A to B, i.e. to transform A to B, an algorithm can be applied:

(1) initialise i = 1, distance = 0 (distance is used to measure distortion);

(2) initialise j = 1;

(3) compare aij with bij

```
if (aij = bij) then
{
        if [(i ≤ m) and (j ≤ n)] then
        {
                j = j + 1
                goto (3)
        }
        else
        {
                if [(i ≤ m) and (j > n)] then
                {
                        i = i + 1
                        goto (2)
                }
}
if (aij ≠ bij)
{
        if [ (i ≤ m) and (j ≤ n)] then
        {
                j = j + 1
                distance = distance + 1
                goto (3)
        }
else
{
        if [(i ≤ m) and (j > n)] then
                {
                        i = i + 1
                        distance = distance + 1
                        goto (2)
                }
}
if [(i > m) and (j > n)] then
        goto (4)
```

(4) The distortion factor is value of the distance.

Inference Matching

Inference matching applies heuristics to derive solutions by similarity identification and comparisons. In contrast to pattern matching, which is mainly based on topological structure, inference matching can be established on feature matching, relation matching, object matching, topological matching and the combination of these matching. All of these are further synthesised as heuristic matching, which forms the centre of this section.

Heuristic matching

Heuristic matching is suitable for a specific group of feature matching problems. The features of these problems are that it is impossible represent the problem to be solved in a way so that every single case is taken into consideration, i.e. it is not practical to represent all cases either because the feature space become too large to implement or the complexity caused by the features' fuzziness increases. The characteristics of these features are: (1) they are used to determine courses of actions based on a **similar** previous example whose solution has been found; i.e. a solution is found by matching the feature of new problem to that of a solved problem rather than finding a solution from a scratch; (2) they cannot be represented and measured in accurate terms. That is to say, these features are either determined on the rules of thumb, or on the structures of problems, or according to the analysis of a particular situation. (3) these features are dynamic and may change if the requirements and environment changes. For example, a database query profile for determining the best access path is dynamically changing, one cannot predict an exact feature pattern for it. In addition, the number of features required can also be dynamic. Depending on the requirement emphasis, say storage utilisation or access speed, the courses of actions could differ. With these characteristics, one may ask, what are the implications of heuristic matching? Heuristic matching is applied to solve problems of the following characteristics:

(1) An expert system where using rules of thumb is efficient: for instance, an inference can be effectively conducted by searching the previous knowledge or successful examples to establish a solution for a new situation, i.e. a new problem P can be guided for a solution by using P', a solved problem. In many situations, feature values can be difficult to decide upon for a specific case. These situations require feature knowledge about their history to be kept. Instrument diagnosis, for example, belongs to this category. In such an environment specific

values of usage period, temperature, pressure, when a fault is likely to occur can be very difficult to decide upon, especially when a number of combinations of these factors can play their role together, significantly different from considering a single factor. However, if a fault is diagnosed, feature information can be collected and stored for the specific moment when the failure occurred and can be used as a reference knowledge for latter diagnosis to prevent the occurrences of similar faults. That is to say, a knowledge-based diagnosis can employ the feature knowledge about the previous instance, matching a current instrument to be monitored, with the stored knowledge of a past similar instance to predict a likely fault.

(2) An expert system where features extraction for representing a possible solution is efficient. An analysis can be extracted from large number of resources for solving a complicated problem. This analysis can be features used for problem-tackling. Analogical reasoning, for instance, can be classified into such a situation, where a solution to a new problem is base on a solution applied to a 'similar' problem. The dominant factors of feature extraction for an expert system relies heavily on features chosen and measurement adopted.

(3) Feature values of a problem to be tackled are dynamic (changing) and difficult to be predetermined or they are dynamically dependent on current environment or situations. In such situation rules cannot be determined by accessing specific feature measures. A set of profiles can provide a changeable framework for heuristic matching.

These profiles are provided in hierarchies: a meta-level which represents what a profile consists of . They are relatively static, and include features as complete as possible. Measurement-level which support feature values.

An overview of heuristic matching

Using heuristic matching to solve a problem P can be described as follows.

(1) Describe P using its extracted features.

Example

An intelligent stock portfolio management system extracts credibility, recommendation grade, industrial, major market, annual sales growth rate, debt ratio, fixed ratio, current values, current ratios, past years' financial results,

export amount, stage of life cycle (electronics, say, increase. Shipbuilding, say, decrease), etc. as feature information. A fault diagnosis system can collect pressure, temperature, environment parameters, etc. as its representative features.

(2) Assign priority to these features based on feature values and domain-oriented knowledge the problem possesses.

Example

A skill match system used for recruitment, for instance, can use the job requirements to assign priorities to various skills of a person. The job requirements may distinguish highly demanded skills, special skills from general skills. Highly demanded skill receives the highest priority to guide the skill match process. Assuming R = {r1, r2, ..., rx} is a set of requirement set, Sy = {Sy1, Sy2, ..., Syz} is a set of skills for person y, here y = 1, 2, ..., n, and n is the number of persons available for the skill match. Employing the knowledge of requirements, the highly demanded skills are firstly matched with each individuals' skill sets. The advantage is to achieve better match performance by eliminating individuals who do not have the highly demanded skills. For different tasks, the determinant factors would be varied.

(3) Find a "similar" problem in the knowledge base P' for P by feature matching, i.e. deploying knowledge of P'.

Example

In a fault diagnosis system, feature values in the previous case can be applied as sample data for monitoring similar situations. These similar situations are Ps and the previous cases are P's. P's are analysed and presented after analysis in the knowledge base and are used as reference features for new situations.

(4) Utilise the solution for P' to guide a solution for P.

Example

In the skill matching process, feedback from previous successful or unsuccessful matching can be analysed by recruitment consultants. The results of this analysis is represented in the knowledge base for reference. In this particular case, the knowledge may include effects of missing skills for certain jobs - both as successful or unsuccessful instances. A programmer's job requirements may add a significant factor to hardware orientation; whereas the

significance may be proved to be irrelevant by large amounts of previous successful match which ignore the hardware orientation, but the computer language skills. In addition, the length of experience in the required fields may prove to be insignificant for a dynamically changing environment, but may be significant for the initiative of facing new challenges with self-learning capability. As a result, newly added features and deleted features (stored as knowledge) improve the success rate of the matching process.

Heuristic matching is employed for problem-solving where it is possible to construct a profile. The profile can be represented at meta-level, which is relatively static, covering as wide as possible for a problem domain; and at the specific measurement level which is determined by individual cases and the rule of thumb and can change during inference process. For instance, in a computer hardware selection process, definition of scale (parameters of main frame, mini-computer, and macro-computer - CPU cycle, size of main memory, size of secondary memory, block size, etc.) changes as hardware advances. The meta-level profile contains a set of specific parameters which are used to describe features of interest. The specific measurement is made up of a set of thresholds describing feature classes of a problem.

The meta-level profile is deployed to find all possible relevant features of a new problem - reflecting the nature of a problem structure. A database, which holds a set of abstract profiles, acts as standards/criteria for a heuristic match with new problems. A heuristic matching inference structure can be illustrated in Figure 4.30.

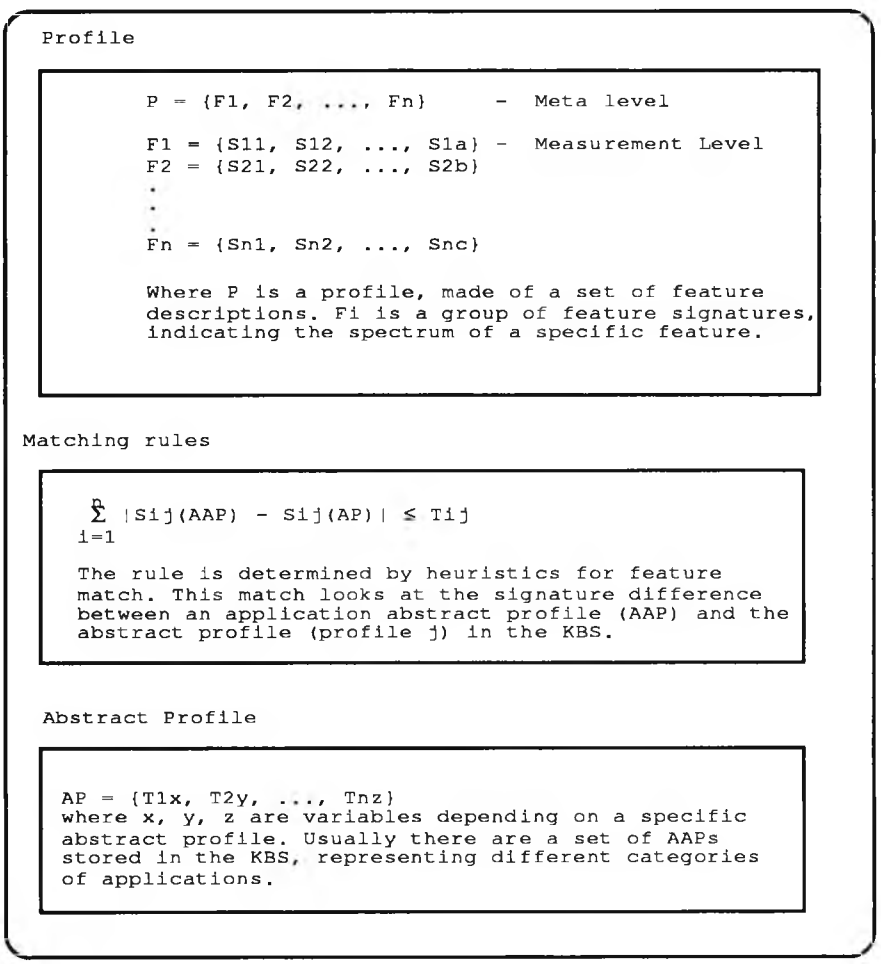


Figure 4.30 Heuristic matching inference structure.

The components of a heuristic matching are: (1) an abstract profile base, which is used to store the meta-level profile and specific measurement; (2) a rule base, containing the similarity rules, incomplete information processing rules, and adjustment rules. From the diagram, it is clear that a successful heuristic matching mechanism must keep the abstract profile base minimal and make the similarity comparison simple and effective. This can possibly be realised by (1) merging common elements of KB to reduce redundancy (high-level abstraction); (2) embedding elimination rules whenever applicable (carefully choosing rule visitation order).

Abstract profile base - design considerations

The practicality of using heuristic matching to solve a problem, is dependent on the complexity of the abstract profile (size and representation). Size determines knowledge complexity; representation decides the inference structure by which paths a solution can be searched, based on known knowledge. To reduce the size of the reference objects - the abstract profile size is the motivation of applying heuristics. The reason is: (1) heuristics allow us to avoid the capture of a complete spectrum of the profile instances, (spectrum represents features). Only information about features at turning point (which influences solution) is recorded. (2) Heuristics facilitate varied solutions to be applied to different situations. Thus the construction of an abstract profile base involves educated guesses of a partition strategy and experience over the feature spectrum. The partition is determined by experience associated with the goals and employed as reference profile instances which the matching process is based on. For example, a feature of a concerned problem can be presented by a limited set of values, measuring the degree/significance of the impact on the problem solution when one of the values is presented.

The heuristic rule consideration

The key to a successful inference system is based on two aspects: accurate data, right knowledge to carry out the inference; and a well-constructed inference strategy. They both depend on acquisition and representation. For a heuristic matching the following criteria are introduced for constructing profiles:

(1) simplicity: simple for similarity identifications.

Simplicity aims at reducing size of features required for similarity identification,

decreasing the steps involved for similarity identification (unnecessary matching paths eliminated), and efforts demanded for constructing a profile for a problem to be solved.

(2) generality: general enough to cover a range of problems (this also reduces the size of the known feature profile set).
Generality depends great deal on selected features, which depicts the profile, the meta-level representation of knowledge. Meta-level knowledge can be domain independent or dependent. The meta-level controls are usually domain independent, whereas, the meta-level features are domain-oriented. Hence a meticulous understanding is the determinant factor for choosing these features. In addition the taxonomy of the domain knowledge ameliorates this process.

(3) simple structure for easy implementation.
This criterion considers how to efficiently implement domain-knowledge by computer representation. In particular this includes inference structures. For example, a tree structure for wave representation is effective because the peaks of wave can match the nodes in the tree structure.

(4) explanatory: matching solution/problem structure for easy explanation.
A complete knowledge base needs to provide adequate explanation facility to users, allowing to ask WHY questions to the system. This implies that the inference steps involved for a specific situation need to be reserved in the system for reference.

A description of the heuristic matching

We introduce the following symbols before we describe the heuristic matching. These symbols are:

HM	- the heuristic matching process, represented by extracted features based on domain knowledge.
APB	- an abstract profile base.
AP	- a set of abstract profile.
F	- a set of names representing concerned features of a profile (meta-level).
P	- a set of instances of the abstract profile, containing its feature information presented by concrete boundary values distinguishing

between various solution categories (measurement level).

V - specific value sets within the spectrum of features, indicating a specific profile instance, measuring a specific instance (feature measurement level).

HR - set of heuristic rules.

Ax - an application(/object /relationship /model/problem) identification

Cx - a set of category information of Ax, depending on measurement level.

Ex - a set of data gained for a specific Ax of interest (measurement for a specific profile instance).

? - a missing value in the spectrum of a feature/measurement. This happens either as a specific value cannot be determined by the time it is required or user does not have any knowledge about it.

S - thresholds to measure the similarity between an abstract profile and a considered profile.

W - the weights given to features in consideration.

Having defined the above symbols we have APB(meta) = {F,AP, HR} at the meta-level:

where F = {F1, F2, ..., Fq},
HR = {HR1, HR2, ..., HRm},
AP = {AP1, AP2, ..., APx};

we then have APB(data) = {P, V, W, S} at the data level:

where P = {P1, P2, ..., Pn},
V = {V1, V2, ..., Vn},
W = {W1, W2, ..., Wq},
S = {S1, S2, ..., Sq};

we also have Ax = {Cx, Ex} at the specific problem level:

where Cx = {Cx1, Cx2, ..., Cxr}
Ex = {Ex1, Ex2, ..., Exq}

in which Exj can have missing value ? for j = 1, 2, ..., q.

The heuristic matching process can be described as:

$$HMi(Vi,Ei) \Rightarrow \min \left(\sum_{i=1}^{n} |Exi - Pi| \right)$$

$$HMi(Vi,Ei) => \min \left(\sum_{i=1}^{n} Wi \, |Exi - Pi| \right)$$

where Wi x |Exi - Pi| = Sxi(Exi, Pi, Wi). Pi is an expected value for selecting a solution x in terms of attribute i, and Exi is an estimated value for problem to be tackled; and Wi is the weight value for feature i. If there are m attributes to be considered, i ranges from 1 to m.

Fuzzy information processing

The objective of employing the fuzzy set concept is to reduce the complexity of the world we are going to represent by introducing a way of representing the vagueness of information. This concept is used where a sharp line is difficult to draw or clarity does not exist. For example, it is difficult to define a degree of clear sky or cloudy sky; it is not helpful to answer the question :" Is it farther in distance from New York City to London or from Washington D.C. to London? " by how many centre meters between New York and London.

Vagueness in describing data distribution, for example, can be described as even, uneven, and relatively even, fairly even, almost uneven and so on. These terms are fuzzy because no clear boundaries can be drawn to distinguish between even and fairly even data distributions. Employing the fuzzy set concept, these fuzzy terms can be defined by a function, Deven, where $0 \leq \text{Deven} \leq 1$. In addition, it is not clear what range of data distribution will match the ability of an access algorithm optimally. In description, we can confidently say that the quantile-hashing algorithm deals with uneven data distribution better than z-hashing algorithm; PLOP-hashing algorithm handles dynamic situations better than quantile-hashing algorithm. However, it is very hard to say in what circumstance an optimal match between an algorithm and an application is achieved because this knowledge has to be gained and built into the expert system by implementing and analysing a number of applications. Furthermore, features of a problem are dynamic, which implies that an optimal initial match will not mean an optimal match during the life cycle of an application. The dynamic features will be dealt with by performance tuning.
To deal with the fuzziness involved in the experimental system effectively, the fuzzy set concept has been introduced to represent features in their spectrums. This representation will be divided into two levels for introducing changes to knowledge

stored for performing heuristic matching, selecting and tuning tasks.
The first level defines the relationship between fuzzy terms and their corresponding fuzzy sets, the second level defines measurement relates to a specific solution. The logical representation can be illustrated in Figure 4.31.

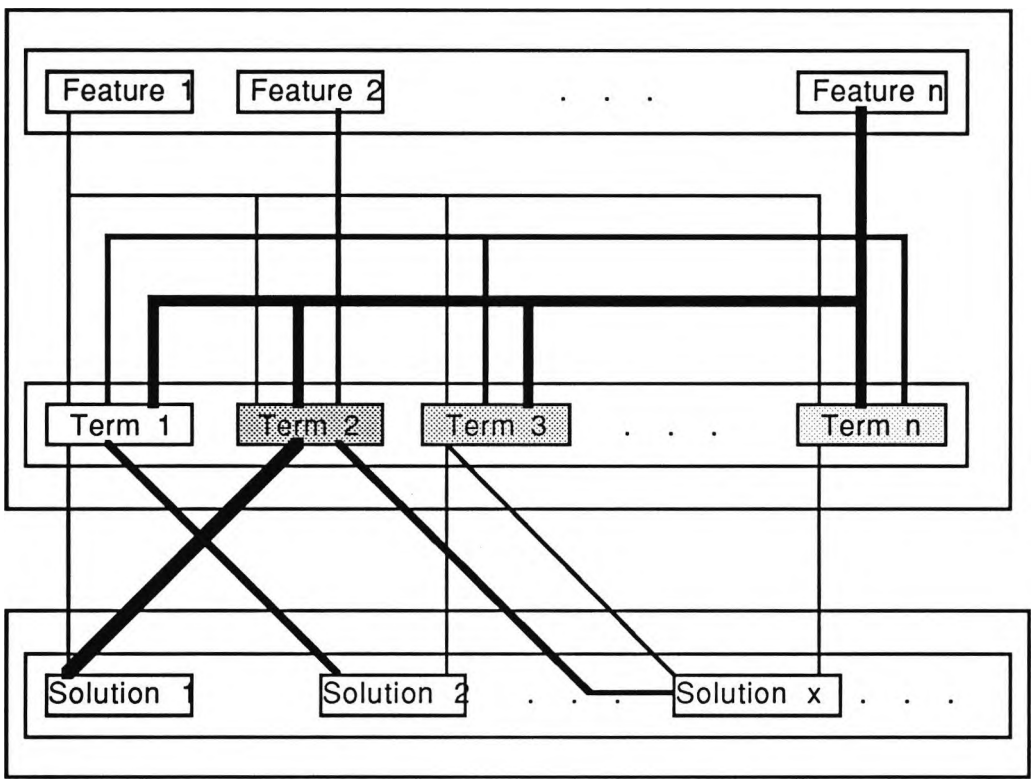


Figure 4.31 Fuzzy Information Logical Representation.

In the diagram, different types of lines represent varied degrees of fuzzy terms. A relationship is established between the spectrum of individual features and a particular solution which prefers a specific range of a feature specified by its value in the fuzzy set.

Since matching is designed to find similarity or dissimilarity among more than two objects, sometimes, similarity can be measured quite accurately. For example, two rectangles can be measured by their length and width against each other; two circles can be measured by their radii against each other. Sometimes, the similarity is difficult to be represented accurately. For instance, two complicated images will be very tricky to be compared either due to the complexity of representing an accurate resolution or due to distortion among one of the images. Furthermore, the definition of similarity is very much objective-oriented: similarity could refer to the minimal difference introduced from one object to the other (an analogical similarity); similarity could allow symmetrical information to occur in different orders or positions from one subject to the other (a reflective similarity); similarity could also refer to structure equality without concerning the size (a topological structure similarity). Heuristic matching is developed to overcome the difficulties of incomplete and fuzzy situations and reduce complexity caused by constructing exact profiles. Rules of thumb are utilised to establish the matching process according to the goals of an application system.

4.1 Heuristic matching - the control strategy

To simulate heuristics, it is important to consider the control strategy. One of the major differences between thinking or solving problems based on rules of the thumb (experience) and based on an algorithmic approach is characterised by the control strategy - heuristic judgement. Applying rules to tackle a problem, the visitation order of rules makes differences in terms of efficiency to arrive at a solution. Moreover, the steps involved in heuristic matching are changeable, i.e. for different situation or requirements, some steps can be skipped (eliminated). An example of this is that, at a certain point of tackling a problem, the salient features of an instance considered, which meets a particular goal (solution), further judgement can be skipped. In other words, the control strategy states the following facts: if path i is satisfied then skip all the rest. Here path i is one of shorter paths to the solution in the inference engine.

When facing a set of feature information belonging to a problem to be considered, what is the best visitation order of an applicable rule set? In this section the answer to this question will be discussed.

What respects should we consider to address the control strategy? The following will be considered before constructing a visitation order.

(1) analysing feature significance based on the value of a signature;
(2) analysing inter-relationship between features;
(3) analysing possibility of excluding visitation for some rules at early stage (eliminating rules).

Based on the analysis, a matching between the results of the analysis and the rule structure are used to determine the visitation order of the rule set. This consideration will allow the system to skip unnecessary parts of the rule set. The process can be shown in Figure 4.32.

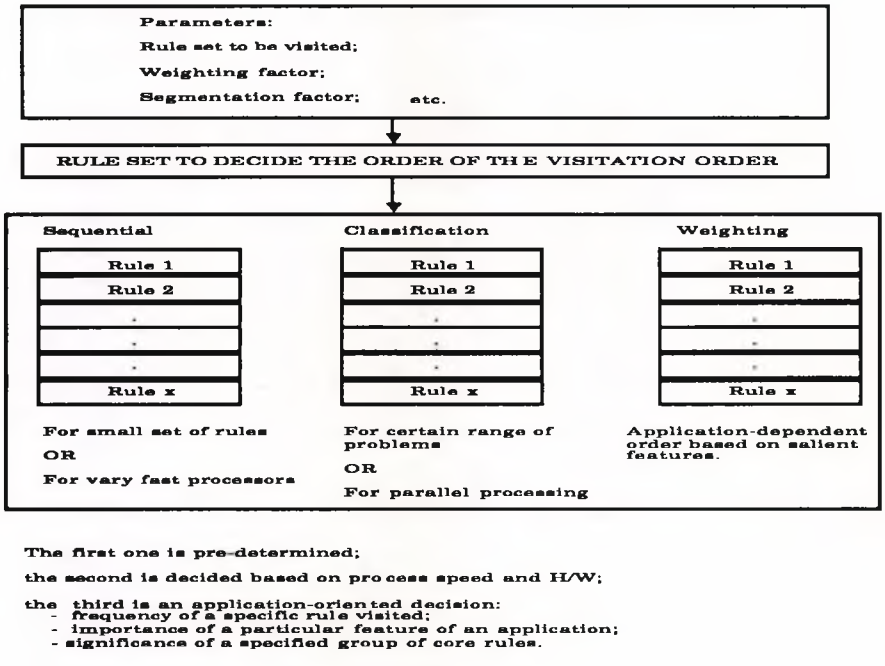


Figure 4.32 Constructing the order to visit the rule set.

For any problems, rules concerning its scale and life span will be examined to determine whether further consideration is required. Following this step the possibility of elimination is further cared for. The salient feature of a problem is considered; paths leading to solutions which fail to satisfy the requirements by important features are eliminated. A high Wi (where Wi is the weight given to attribute i) indicates that rules relating to attribute i will be visited firstly. If one of the chosen abstract profiles matches the problems of interest the solution used for this profile is selected. If none of the other attributes are matched with the these profiles, the visitation of the attribute j with the next highest Wj is triggered.

Searching considerations

Search consideration concerns two issues: (1) the physical data structure for storing knowledge needed to tackle a range of problems; (2) a set of search algorithms that can be effectively applied to this structure to cope with changes, and to achieve necessary accuracy. The first issue depends on characteristics of a problem in hand; the second issue is analogical to physical database design.

Much literature in A.I. has been concerned with search strategies [RI88]. Heuristic approaches are also considered. Similar to any techniques for problem solving, the characteristics of problems are important factors to consider. A problem, which requires a large body of knowledge to tackle it, is favourable in the heuristic search approach (in order to achieve a near-optimal solution); the solution to a problem, that needs great accuracy, may introduce intensive searches to achieve that goal using the cluster technique, which gathers similar features together, and can reduce the efforts of an intensive search.

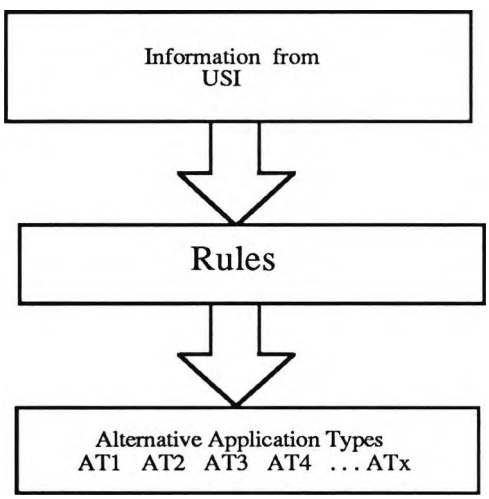
Conclusion

Matching techniques are widely used in information systems. A production system employs the matching concept to derive solutions from LHS and working memory; an image processing deploys the matching concept to compare a predetermined template with images to be recognised; a pattern recognition uses the matching concept to find similarities and distortions of a criteria and a pattern, using heuristics as part of the mechanism.

4.4.1. Initial algorithm selection

To compare an application with an AAP the performance stored in the DBP is transformed into the same scale and categories of the AAP. In sections 3.2.1. and 3.2.3. an initial database profile from USI and a complete database profile DBP are defined respectively regarding the data and query features, environment constraints and application performance. An AAP concerns data distribution, query features and performance. The rules, which perform selection of an algorithm by application classifications, are introduced in several forms: initial algorithm selection rules, similarity comparison rules and heuristic decision rules. They aim at facilitating the selection of algorithms.

An initial database profile established by the system through the USI, provides information by itself, to identify its category. In section 3.2.1. the USI has been defined in terms of data requirements and environmental constraints. How do we use the information to recognise the characteristics of an application? The rules constructed here will guide the system to make use of the information gained from USI. The framework is pictured as:



In the AAP base each application type is given a set of values for a number of parameters. They are used as the basis for a similarity comparison. To accommodate incomplete information supplied to the system, data about various applications needs to be collected and analysed. The results of the analysis are stored in the form of an AAP in the system. These AAP profiles are classified according to their application areas (applications with similar features are categorised as the same area, i.e. their features fall in the same range of measurement) and are used as a source of incomplete information supply for an initial selection of an algorithm. If a desired AAP, which is within the same application class, cannot be found in the system then the system will make assumptions to complete the missing data. These assumptions may be inaccurate so that the dynamic tuning and monitoring process will take care of it later. As this application develops it can gradually be formed as a new AAP profile being added to the system. The system mainly deals with two levels of tuning:

(1) static level - initial selection of an implementation algorithm;
(2) dynamic level - monitoring salient performance changes to the database.

The information concerned here is dimensionality m, resolution level r, bucket size b and information about data distribution. The values of m, r, and b are decided by the application search space and data set size. Other information is derived from partitioning the data space. As described before C[i] for i = 1, 2, ..., r are stored by the system. The information in a useful form will then be derived from the obtained data through the USI. It is described below.

$$r = \left\lceil \frac{n \times (1 + Ir)}{b} \right\rceil$$

r is used as an estimated resolution level for partitioning the data space.

data distribution: data distribution relates to C[i] for i = 1, 2, ..., r from which a set of parameters (d(even), Nover, Nempty, p, Ld(si) for i = 1, ..., m, j = 1, ..., sj can be derived. They are defined in Appendix A7.

At this initial stage information is obtained from the USI. The application is not compared with the AAP for simplicity. The AAP, if found, is used to derive missing information. This is done by searching the AAP base for an ATx (see the

example in section 4.8.). If no ATx is found the system will generate a sample data space by a given n to derive required information under some assumptions, such as resolution r and a given bucket size b. Other alternative ways of deriving missing information can also be introduced. As required information is supplied the rules will classify an application according to an applicable algorithm. Each rule will have an explanation text associated with it. During the process the values of each parameter used for reasoning are recorded in a system template to assist the reasoning explanation. This will be discussed shortly.

Eliminating rules

A rule set should be constructed based on heuristic knowledge, which can make the reasoning simple. Making decisions on heuristics, a search path for a solution can be formed by using the most dominant factors (called salient factors) if they are known. For instance, considering physical database design, if the access time is an important factor to be considered and the size of a data set is large then the first rule set Rset1 (rules for selecting the EXCELL algorithm) can be skipped. The search for a solution thus uses the idea of elimination to skip unnecessary rule searches that are not likely to support the application. We classify these rules as 'eliminating rules'. Employing the same principle, matching working memory with the LHS can be implemented by given a set of salient factors, resulting in a dynamic sequence of rule match.

The eliminating rules:

ER1 = { |index| ≤ M --> ALT[1] }
ER2 = { (tm ≤LS) --> ALT[1] }
ER3 = { ((Td ≥ 2 x Tsec) and (d(even) < d2)) --> ALT[1] }
ER4 = { ((d(even) < d1) and (Dyn < dy1)) --> ALT[2] }
ER5 = { ((d(even) < d2) and (Rs < r1)) --> ALT[3] }
ER6 = { (Dyn > dy2) --> ALT[4] }
ER7 = { ((Dyn > dy2) and (d(even) > d3)) --> ALT[5] }
ER8 = { ((Dyn > dy2) and (Ds = {obji for i = 1, ..., n}) --> ALT[6] }

Initially we set:

M = 64K, LS = 1 week
d1 = 25%, d2 = 30%, d3 = 40%
dy1 = 10%, dy2 = 50%
r1 = 50%

i.e. we translate these rules as:

ER1 = { |index| ≤ 64K --> ALT[1] }
ER2 = { (tm ≤ 1 week) --> ALT[1] }
ER3 = { ((Td ≥ 2 x Tsec) and (d(even) < 30%)) --> ALT[1] }
ER4 = { ((d(even) < 25%) and (Dyn < 10%)) --> ALT[2] }
ER5 = { ((d(even) < 30%) and (Rs < 50%)) --> ALT[3] }
ER6 = { (Dyn > 50%) --> ALT[4] }
ER7 = { ((Dyn > 50%) and (d(even) > 40%)) --> ALT[5] }
ER8 = { ((Dyn > 50%) and (Ds = {obji for i = 1, ..., n}) --> ALT[6] }

As new knowledge is added to the system the values can be altered.

The meaning of these rules can be interpreted as:

Rule ER1

IF: |index| ≤ M

THEN: ALT[1]

REASON: if the index file can be stored in main memory then choose ALT[1] (the EXCELL algorithm)

Similarly for the other rules.

These eliminating rules are constructed based on the idea that the conditions (premises) spell out the critical factors that an algorithm (conclusion) is likely to be applied to its best advantage.

For each eliminating rule there is an attached explanation:

EER1: since the index file size calculated according to the partition PT1 is small the EXCELL algorithm is chosen and no further judgment is required.

EER2: since the life span is short (determined by the value of LS) we choose EXCELL algorithm.

EER3: since more than one secondary device access is allowed, and the data distribution is relatively even, the EXCELL can be chosen without further consideration.

EER4: since the data distribution is even the z-hashing can be selected and all other rules are skipped.

EER5: since quantile-hashing can deal with a dynamic situation reasonably well and it preserves the geometric proximity as the z-hashing it is selected without further consideration.

EER6: since the insertion and deletion rates are high the PLOP-hashing is

chosen without further consideration.

EER7: since the insertion and deletion rates are high and the data distribution is non-uniform the BANG file algorithm is chosen and no further rules need to be applied.

EER8: since R-tree can cope with an object database well, especially when the size of the objects and the database may alter, it is chosen for a dynamic object database without further consideration.

The eliminating rules do not necessarily provide the best solution in terms of performance. They are used as the first set of rules for initial algorithm selection, where the considered application has a specific feature that matches a particular access algorithm, because an algorithm which provides the best performance may not offset the effort.

Initial algorithm selection rules

ATR = {ATR1, ATR2, ..., ATR6}
= {EXCELL, z-hashing, quantile-hashing, PLOP-hashing, BANG file, R-tree}

Rules for assigning weight for initial algorithm selection

There are four conditions to be examined; each one can be assigned a value from { 0, 1, 2, 3, 4, 5 } according to its significance. The corresponding significance can be translated as {0 --> not concerned, 1 --> not important, 2 --> little importance, 3 --> leave to the system to decide, 4 --> important, 5 --> very important }. The following WRi for i = 1, 2, 3, 4 demonstrates the meanings of these assigned values. More values can be introduced to represent refined degree of various features.

WR1 = {	memory size is not concerned	--> w3 = 0,
	not important	--> w3 = 1,
	of little importance	--> w3 = 2,
	leave to the system to decide	--> w3 = 3,
	important	--> w3 = 4,
	very important	--> w3 = 5
WR2 = {	expected storage utilisation is not concerned	--> w1 = 0,

not important --> w1 = 1,
of little importance --> w1 = 2,
leave to the system to decide --> w1 = 3,
important --> w1 = 4,
very important --> w1 = 5 }

WR3 = { ability to deal with dynamic situation not concerned
--> w2 = 0,
not important --> w2 = 1,
of little importance --> w2 = 2,
leave to the system to decide --> w2 = 3,
important --> w2 = 4,
very important --> w2 = 5 }

WR4 = { fast range retrieval not concerned --> w4 = 0,
not important --> w4 = 1,
of little importance --> w4 = 2,
leave to the system to decide --> w4 = 3,
important --> w4 = 4,
very important --> w4 = 5 }

Note that the weight rules are mainly stored for associating meanings with each weight value.

ATR1 - application type which chooses EXCELL as the implementation algorithm.

ATR2 - application type which chooses the z-hashing as the implementation scheme and so forth.

ATRA1 = { if status = 'missing' then action-1 }.
The rule says that if information is missing then execute action-1, which selects an AAP from the system and fills in the missing information for an application.

ATRA2 = { if priority = 'y' then action-2 }.
The rule tells us that if the user's requirements are to be considered then execute action-2 assigning weights to different conditions.

ATR1 = { (|index| ≤ M),

(d1 < d(even) < d2),
(dy2 < Dyn < dy3),
(r2 < Rs < r3) --> ALT[1],
(|index| > M) & (Td = 2 x Tsec),
(d1 < d(even) < d2),
(dy2 < Dyn < dy3),
(r2 < Rs < r3) --> ALT[1] }

ATR2 = { (|index| > M),
(d(even) < d1),
(Dyn < dy1) (Rs > r5) --> ALT[2]}

ATR3 = { (|index| > M),
(d2 < d(even) < d3),
(Dyn < dy1), (Rs > r5) --> ALT[3]}

ATR4 = { (|index| > M),
(d2 < d(even) < d3),
(Dyn > dy5), (r1 < Rs < r2) --> ALT[4] }

ATR5 = { (|index| > M),
(d(even) > d4),
(dy3 < Dyn < dy4),
(Rs > r5) --> ALT[5] }

ATR6 = {(Ds = {obj1, ..., objn}) and (Dyn > dy5) -> ALT[6] }

Initially we set:

Deven = { d1, d2, d3, d4 }
= { 20%, 25%, 40%, 50% }

DY = { dy1, dy2, dy3, dy4, dy5 }
= { 10%, 20%, 30%, 40%, 50% }

RS = { r1, r2, r3, r4, r5}
= { 10%, 20%, 30%, 40%, 50% }

i.e. initially the rules are translated as:

ATR1 = { (|index| ≤ M),
(20% < d(even) < 25%),
(20% < Dyn < 30%),
20% < Rs < 30%) --> ALT[1];
(|index| > M) & (Td = 2 x Tsec),
(20% < d(even) < 25%),

			(10% < Dyn < 30%),	
			(20% < Rs < 30%)	--> ALT[1] }
ATR2	=	{	(|index| > M),	
			(d(even) < 20%),	
			(Dyn < 10%),	
			(Rs > 50%)	--> ALT[2] }
ATR3	=	{	(|index| > M),	
			(25% < d(even) < 40%),	
			(Dyn < 10%),	
			(Rs > 50%)	--> ALT[3] }
ATR4	=	{	(|index| > M),	
			(25% < d(even) < 40%),	
			(Dyn > 50%),	
			(10% < Rs < 20%)	--> ALT[4] }
ATR5	=	{	(|index| > M),	
			(d(even) > 50%),	
			(30% < Dyn < 40%),	
			(Rs > 50%)	--> ALT[5] }
ATR6	=	{ (Ds = {obj1, ..., objn}) and (Dyn > 50%)		--> ALT[6] }

Using d(even), DY and RS to represent the boundaries allows us to change their values without affecting rules.

The meaning of these rules can be interpreted as:
Rule ATR1:

IF:	(|index| ≤ M)	or
	(20% < d(even) < 25%)	or
	(20% < Dyn < 30%)	or
	(20% < Rs < 30%)	
THEN:	select ALT[1] = EXCELL	

REASON: if the index can be stored in main memory and the data distribution is reasonably even or the data set is not volatile or the range search rate is not high then the EXCELL algorithm is chosen. Similarly for the other rules.

Obviously, several conditions may be missing and different emphasis may be

added to different factors. To determine which algorithm should be chosen we evaluate all six rules to see which is optimal. This is done by constructing a function h(ATRi). An example will be given in section 4.8 to show how a choice is made by the six rules.

For each given rule the explanation text is given and will be shown in response to a user's request and the reasoning process is stored in a template. For this set of rules, the values used to calculate h(ATRi) are kept. These values are used to extract relevant explanations.

Explanation

explanation 1 for ATR1

EXCELL algorithm is selected

(a) the first rule is used as the conclusion,

(b) the second rule is used in the selection of the algorithm,

EX1.1 (a) the index file can be stored in main memory so that the algorithm is selected for its simplicity,

EX1.2 (b) the index file cannot be stored in main memory, but the access time requirement allows two secondary storage accesses,

EX1.3 the data distribution is moderately even so that the index entries for empty grid cells may offset the holes in the data file,

EX1.4 the insertion and deletion rate is moderate and therefore the reorganisation is done mainly within the index file,

EX1.5 the range search rate is moderate since the proximity is only guaranteed for the index file and not for the data set itself.

explanation 2 for ATR2

The z-hashing algorithm is selected

EX2.1 the index file estimated for the EXCELL algorithm will exceed the capacity of main memory so that an extra access is required for a point search if the EXCELL algorithm is chosen,

EX2.2 the data distribution is even. This implies that fewer data holes will be introduced by the z-hashing,

EX2.3 the insertion and deletion rate is low so that the possibility of deterioration and reorganisation is reduced,

EX2.4 the algorithm has good performance for the range search.

explanation 3 for ATR3

The quantile-hashing algorithm is selected

EX3.1 = EX2.1,

EX3.2 the data distribution is relatively uneven and the algorithm allows controlled partitioning of the data space,

EX3.3 = EX2.3,

EX3.4 = EX2.4.

explanation 4 for ATR4

The PLOP-hashing algorithm is selected

EX4.1 = EX2.1

EX4.2 = EX3.2

EX4.3 the data set is highly dynamic (the insertion and deletion rates are high) and the algorithm caters particularly for such a situation

EX4.4 the range search rate is relatively low as the flexibility of coping with a dynamic situation may cause the loss of data proximity in the storage of data

explanation 5 for ATR5

The BANG file algorithm is selected

EX5.1 = EX2.1

EX5.2 the data distribution is uneven and the algorithm divides the data space into m-d cubes which minimise the performance deterioration caused by an uneven data distribution

EX5.3 by redistribution of data items the chosen algorithm deals with a dynamic situation reasonably well

EX5.4 = EX3.4

explanation 6 for ATR6

The R-tree algorithm is selected

EX6.1 = EX2.1

EX6.2 = EX1.3

EX6.3 = EX1.4

EX6.4 = EX3.4

Since each rule includes a multitude of conditions, and it is not realistic that all of them can be satisfied at the same time, a function is introduced to evaluate the "condition matching" degree of each algorithm. This can be done by assigning weights to each condition according to performance requirements. The above rules have four conditions. According to the requirements of an application different weights can be assigned to each of them, reflecting the priority of conditions. The weighted values are formed flexibly with regard to the nature of an application and performed by action-2 through to rules for assigning the weights.

An example

Suppose the information gained from USI is ATR = { lindexl > M, d(even) > 50%, Dyn > 50%, Rs > 50% }. The action-2 assigns the following weights to each condition: W = { w1, w2, w3, w4 } = { 1, 1, 5, 1 }. To make a judgment of applying these rules a heuristic function h(ATRi) for i = 1, 2, 3, 4, 5, 6 is constructed.

$$h(ATRi) = \sum_{i=1}^{i=4} xi$$

$$\text{where } xi = \begin{cases} 0 & \text{if the condition is not satisfied} \\ wi & \text{otherwise} \end{cases}$$

the weight wi is assigned according to the requirements and using rules WRi.

If max(h(ATRi) for i = 1, 2, ..., 6) = h(ATRx) then ALTx is selected as the implementation algorithm. For the data given above we get:

h(ATR1) = 0 + 0 + 0 + 0 = 0
h(ATR2) = 1 + 0 + 0 + 1 = 2
h(ATR3) = 1 + 0 + 0 + 1 = 2
h(ATR4) = 1 + 0 + 5 + 1 = 7
h(ATR5) = 1 + 1 + 0 + 0 = 2
h(ATR6) = 1 + 0 + 0 + 1 = 2

max(h(ATRi) for i = 1, 2, ..., 6) = h(ATR4) and therefore ALT4 = PLOP-hashing is selected for the application.

Explanation

All of the above explanations are given under the condition that h(ALRi) is not equal to zero. In the example, when ALT[4] is selected we can represent h(ALRi) as a matrix form and store the matrix for explanation.

j =		1	2	3	4	
i =	1	0	0	0	0 = 0	
	2	1	0	0	1 = 2	
	3	1	0	0	1 = 2	
	4	1	0	5	1 = 7	<-- ALT[4] is the choice
	5	1	1	0	0 = 2	
	6	1	0	0	1 = 2	

The explanation numbered ij where xj is of non-zero value and ALT[i] is the chosen algorithm to be shown, i.e. if an explanation for the PLOP-hashing is requested then the following information will be illustrated:

the PLOP-hashing algorithm is the choice

x1 = 1

EX2.1 the index file estimated for the EXCELL algorithm will exceed the capacity of main memory so that an extra access is required for a point search if the EXCELL algorithm is chosen.

Storage utilisation is of little importance.

x2 = 0

x3 = 5

EX4.3 the data set is highly dynamic (the insertion and deletion rates are high) and the algorithm caters particularly for such a situation. Dealing with a dynamic situation is very important for the application (since x3 = 5)

x4 = 1

EX4.4 the range search rate is relatively low as the flexibility of coping with a dynamic situation may cause a loss of data proximity in storage

The ability to deal with the range search is of little importance.

If ATR = { |index| > M,
d(even) > 50%,
Dyn > 50%,
30% < Ps < 40%,
Rs > 50% } then we have:

$$h(ATRi) = \sum_{i=1}^{i=4} xi$$

$$\text{where } xi = \begin{cases} 0 & \text{if the condition is not satisfied} \\ wi & \text{otherwise} \end{cases}$$

the weight wi is assigned according to the requirements and using rule WRi.

If max(h(ATRi) for i = 1, 2, ..., 6) = h(ATRx) then ALTx is selected as the implementation algorithm(s). For a data set given we get:

h(ATR1) = 0 + 1 + 0 + 0 = 1
h(ATR2) = 1 + 0 + 0 + 1 = 2
h(ATR3) = 1 + 0 + 0 + 1 = 2
h(ATR4) = 1 + 0 + 1 + 0 = 2
h(ATR5) = 1 + 0 + 1 + 0 = 2
h(ATR6) = 1 + 0 + 0 + 1 = 2

max(h(ATRi) for i = 1, 2, ..., 6) = h(ATRj) for j = 2, 3, 4, 5, 6 and therefore, further decisions have to be made. At this point the system can ask the users whether they wish to choose an algorithm by themselves or not. The choices given to the users will help to narrow the solution space. If no choice is given then the system will select the one with the simplest implementation. For an initial algorithm selection the choices will be given in the following manner:

Please choose an algorithm you think the best according to the features given, which may cause a problem:

ALT[2]: Cannot deal with dynamic situation well,
cannot deal with unevenly distributed data well
implementation is more costly than ALT[1].

ALT[3]: Cannot deal with dynamic situation well
implementation is more costly than ALT[2].

ALT[4]: Cannot deal with a range search well
implementation is more costly than ALT[3].

ALT[5]: Implementation is more costly than ALT[4].

4.4.2. Selecting an algorithm by a similarity comparison

Similarity comparison assumes that database applications can be classified into various categories by features affecting physical database organisation. Those features are used to derive a solution based on analogical reasoning. The reasoning process assumes that if a defined similarity can be identified between a new application (AP) and an existing application abstract profile (AAP), the access algorithm applied by the AAP can be selected for the AP.

As a database profile is established gradually at different stages of an application, a complete set of data about a database profile may not be available at the time of the similarity comparison. In addition, different applications emphasise different features and therefore, the system should allow flexible classification of an application based on the available information of a database profile. Furthermore, some parameters such as data distribution and performance are important factors in the selection of an algorithm. Hence they can be used to simplify tuning of a physical database by a weighting function.

The AAPs provide information about different characteristics of various data spaces as criteria for the system. The system classifies applications into different groups so as to give a basis for recognising the category of a current data set. If the similarity between an application and an AAP has been identified, all knowledge stored about the AAP such as the chosen implementation algorithm and the evaluated performance can be used to guide our decisions to a near-

optimal storage of a data set. To recognise the similarity between an application and an AAP the relevant features need to be defined and quantified.

We have chosen several implementation algorithms in our system. To carry out the task of selecting these algorithms the criteria for classifying applications will be considered in terms of parameters which influence the performance.

The rationale for selecting an algorithm by the similarity comparison is to apply an algorithm successfully used for a similar application before a new one. Hence the process is to make a comparison between an application and an AAP and use the algorithm from an AAP upon a satisfied similarity degree. The process can be described briefly as:
S(x, ATx) --> ALT(ATx)
S(x, ATx) -similarity degree for application x compared with ATx in the AAP.
ALT(ATx) -the algorithm used for ATx.

In this section a set of parameters and rules are given for a similarity comparison. A similarity function s(x) is introduced to measure the result.

To conduct a similarity comparison we briefly review the AAP.

Application abstract profile

AAP = { ACi for i = 1, 2, ..., x }

Application class

definition level

ACi = { application area, definition }

detailed level

ACi = { ATi, ADi, AQi, PEi }

Data distribution

ADi = { m, r, b, Ddis }

Query type

AQi = (Qi, F, Ir, Dr, Ps, Rs)

Performance evaluation

PEi = (T, Su, ALT[i])

As an AAP is used as an image a database profile has to be of the same form as an AAP.

Transform a database profile to the form of AAP

An estimation on resolution of the data set:

$$r(u) = \frac{n(u)}{b}$$

$$r = \frac{n}{b}$$

$$\lceil \log_2 r \rceil \quad \lceil \log_2 r(u) \rceil$$

if $2 = 2$ then these parameters calculated from C[i] for i = 1, 2, ..., r can still be used for the data set in the DBPi, otherwise we may decide to introduce a new resolution level. As the database profile changes during database operations, alterations are made accordingly. If a data set is given previously then r is determined and C[i] are used for recording the data distribution. When an insertion or a deletion is made we will store the detail in the system log file or in some other way in order to update the data distribution information kept for the database profile. Similarly searches or retrievals will be recorded by the system to update query information stored for the database profile. An insertion will increase one to one of the C[i] for i = 1, 2, 3, ..., r of the data set; a deletion will decrease one by one of the C[i] for i = 1, 2, ..., r; keeping the Ddis data up to date. When there is a need to compare the database profile and an AAP the required information about data distribution can be deduced -Nover, Nempty, N(d), p and so forth. They are compared with a chosen AAP by the similarity comparison rules. Since with the same resolution the relative bucket size could be different from one data set to another a scale transformation is performed before the comparison. Alternatively, data is collected only at a critical point where a significant performance change has occurred.

After choosing an application type ATi from the AAP we will view the application type as an image and the application as an object. The similarity comparison is carried out between these two and we shall use " ' " to represent data from an AAP in the knowledge base.

Transform DBP to AAP scale before similarity comparison

$$C(DBP)[i] = C(AAP)[i] \times \frac{b(AAP)}{b(DBP)} \quad \text{for } i = 1, 2, ..., r$$

where C(DBP)[i] is the value obtained from DBP
C(AAP)[i] is the equivalent DBP value in AAP scale

$\frac{b(AAP)}{b(DBP)}$ is an image factor.

after this scale transformation we can use C[i] for i = 1, ..., r to represent the values gained from AAP for similarity comparison.

Deviation of data distribution

$$DD = \frac{\sum_{i=1}^{i=r} | C(o)'[i] - C(o)[i] |}{r \times b}$$

where C'(o)[i] and C(o)[i] represent C'[i] and C[i] in an ordered sequence.

Deviation for even data degree

$$ED = \frac{| d'(even) - d(even) |}{r}$$

Deviation for the number of overflow grid cells

$$OV = \frac{| N'over - Nover |}{r}$$

Deviation of dynamic factor

$DF = |Dyn - Dyn'|$

Deviation of point search rate

$DP = |Ps - Ps'|$

Deviation of range search rate

$DR = |Rs - Rs'|$

Deviation for the number of empty grid cells

$$OE = \frac{| N'empty - Nempty |}{r}$$

Deviation for local data densities

$$LD = \frac{\sum_{x=1}^{x=sx} | Ld'(x) - Ld(x) |}{sx \times b} + \frac{\sum_{y=1}^{y=sy} | Ld'(y) - Ld(y) |}{sy \times b}$$

Deviation for query frequencies

$F = \{ fx1, fx2, ..., fxm \}$

$F' = \{ f'x1, f'x2, ..., f'xm \}$

F, F' are ordered frequency functions for the application and an AAP respectively.

$$FD = \sum_{i=x1}^{i=xm} |f'i - fi|$$

For the similarity comparison the above parameters are assessed on a set of given thresholds. Similarity is measured by the degree of deviations defined by

the following rules.

Similarity comparison rules

SR1 = {DD > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%,≤ 50%}
SR2 = {ED > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}
SR3 = {OV > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}
SR4 = {OE > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}
SR5 = {LD > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}
SR6 = {FD > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}
SR7 = {DF > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}
SR8 = {DR > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}
SR9 = (DP > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%, ≤ 50%}

A similarity function is constructed according to the actual values gained from the above deviation range and an AAP. For each range in the rule we assign values corresponding to its meaning and measure the degree of similarity. This is denoted by s(SRi) = { 0, 5, 4, 3, 2, 1 } i = 1, 2, ..., 8. The similarity function is defined as:

$$s(x) = \frac{\sum_{i=1}^{i=m} si(x)}{m} = \frac{\sum_{i=1}^{i=m} \frac{s(SRi)}{mi}}{m}$$

m -the number of rules (m = 9)
mi -number of boundaries defined with non-zero value in each rule
s(x) indicates the degree of similarity in terms of the average percentage to which the application agrees with a chosen AAP. Its value is between 0 and 1. Each rule gives boundaries to measure the difference between two parameters of an ATx in the AAP and an application values represented by s(SRi) for the result. The greater the value of s(SRi) the closer an application approaches the chosen ATx.

To explain the similarity degree assigned by s(SRi) = { 0, 5, 4, 3, 2, 1 } we

use SR1 as an example:

SR1 = {DD > 50%, ≤ 5%, ≤ 10%, ≤ 20%, ≤ 25%,≤ 50%}

s(SR1) = { 0, 5, 4, 3, 2, 1 }

IF: DD > 50%

THEN: s(SR1) = 0 s1(x) = 0

REASON: if the deviation is greater than 50% then the similarity is identified as 0 (insignificant similarity).

IF: DD ≤ 5%

THEN: s(SR1) = 5 s1(x) = 1

REASON: if the deviation is less than 5% then the similarity is identified as 5 (significant similarity).

IF: DD ≤ 10%

THEN: s(SR1) = 4 s1(x) = 0.8

REASON: if the deviation is less than 10% then the similarity is identified as 4.

IF: DD ≤ 20%

THEN: s(SR1) = 3 s1(x) = 0.6

REASON: if the deviation is less than 20% then the similarity is identified as 3.

IF: DD ≤ 25%

THEN: s(SR1) = 2 s1(x) = 0.4

REASON: if the deviation is less than 25% then the similarity is identified as 2.

IF: DD ≤ 50%

THEN: s(SR1) = 1 s1(x) = 0.2.

REASON: if the deviation is greater than 50% then the similarity is identified as 1.

Since the comparison is done in the order listed in the SRi, DD ≥50% will be exclusive.

Several AAPs may be employed for a similarity comparison. The implementation algorithm that matches the one with the maximum value of s(x)

will be chosen as an image for a given application. Performance measured and the s(x) will be kept as the system knowledge to guide better search paths for the rule base. For instance, if $s(x) > 0.5$ and the performance measured for a given set of applications according to the AAP is satisfied for more than 0.8 of chances, then $s(x) = 0.5$ can be used as a threshold to eliminate further comparison. As soon as a degree of $s(x) > 0.5$ is found the search for the AAP can stop. The reasoning process for an algorithm chosen by a similarity comparison is a pattern recognition process. The decision is made by identifying the way a similar application was processed in the past and the process is used to deal with the new situation.

If the user asks why an algorithm was chosen then the system will show the similarity degree, the ACi chosen for the application and the application class including application area and definition, application type. The system can also give details about the properties of the ACi if required. This is done by searching both the recorded similarity and the AAP base. The logical construction of the knowledge base is shown in Figure 4.33. It illustrates the relationship between a conclusion, its reasoning structure, the facts and the heuristics used to perform the reasoning.

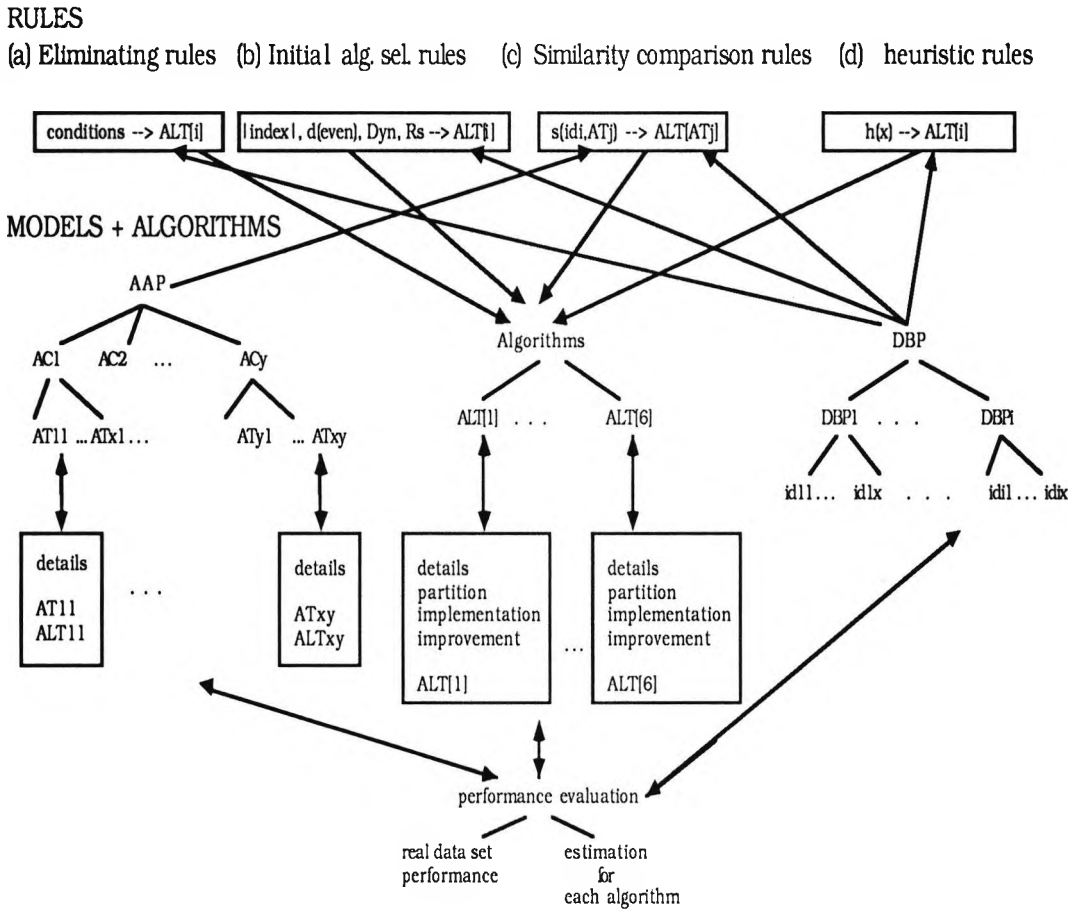
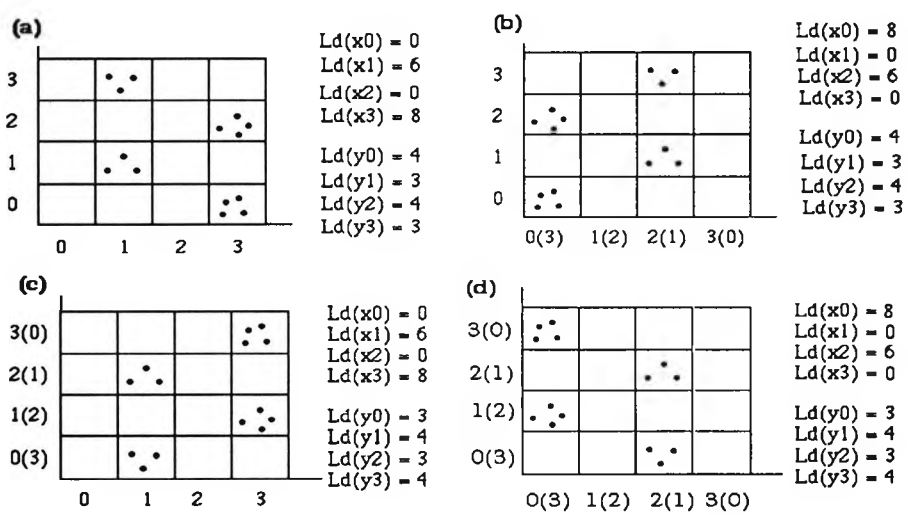


Figure 4.33. The logical structure of the knowledge base.

The symmetric patterns consideration

To organise data we consider that the symmetric patterns are the same in terms of similarity measurement concerned and therefore, in the AAP we only store one type of each pattern once. A transformation from the real data set into the form represented in the AAP is performed before the similarity comparison. An AAP stores a pattern according to its accumulated local data density values, from the lowest slice number to the highest, in the descending order of the value of local data density. An application with a different local density order will be transformed by changing the sequence of the slice numbers from ascending to descending order. A 2-d data space with Sx = (s1, s2, ..., sx) and Sy = (s1, s2, ..., sy) can be transformed into S'x = (s'1, s'2, ..., s'x) where s'1 = sx, s'2 = s(x-1), ..., s'x = s1 and the z-code is formed by S'x and S'y to perform the similarity comparison. An illustration for all possible transformations is shown in Figure 4.34.



For simplicity, Ld is calculated without being divided by ($b \times s_x$) or ($b \times s_y$).
(a) (b) (c) (d) are regarded as the same pattern for their symmetrical properties.

Figure 4.34. Symmetric patterns.

Selecting algorithms by similarity comparison is based on the idea that application features such as data distribution and queries are difficult to describe by a mathematical equation or a mathematical model. In addition, a number of database applications are similar in nature and structures that they should be able to be dealt with in a similar way for optimum efficiency. Moreover, as the number of applications through the system increases better heuristics and measurements can be obtained.

4.4.3. Algorithm selection by heuristics

To select an implementation algorithm several factors are involved.

(1) The distribution of data.

The distribution of data influences the selection of partition strategy. A good partition will achieve a better storage utilisation. The data distribution is of a dynamic property. We use a parametrised method to measure data distribution. We abstract Nempty, Nover, etc. to represent the data features. The partition strategies have been described in section 4.2.

(2) The access mode

The access mode can be measured by the percentage of data accessed in a run or in a query or by the users' specification of application types whether it is real-time or not. It reflects the volatility of the data usage. The calculation on what kind of data organisation is to be used is described in Appendix A9.

(3) The query pattern

The query pattern is measured by frequency function F, insertion rate, deletion rate and search rate. Different partitions, which are geared towards different attribute frequencies, can reduce the physical distance between relevant data so as to optimise the performance as a whole. The algorithm has been given in section 4.3.

In section 4.4.1, we have discussed various rules used for different implementation algorithms separately. To make use of these rules, the priority and the search paths need to be decided. As shown in the decision grid (Table 1). different algorithms vary in implementation complexity. The table is shown in an increasing order of complexity. The algorithm appearing at the last entry of Table 1 tends to be more difficult to realise. Bearing this in mind, we aim to choose the easiest one if the requirements can be met. Therefore the search strategy is selected considering

this factor. The table represents heuristic ideas to match properties of an application to an available algorithm. To implement the idea we construct a heuristic function h(x), which is a function of selecting an algorithm based on the properties of an application. The heuristic function is used when there is no existing chosen ATi in the AAP that matches the application.

possible case ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30		m
tm ≤ 1wk	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	...	
tm ≤ .5yr		Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	...	
n < 10K		Y	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	...	
n < 50K														Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	...	
dis≤20%			Y	N	N	N	Y	N	N	N	N	N	N	Y	N	Y	N	Y	N	N	N	N	N	Y	Y	Y	Y	Y	N	N	...	
dis≤40%					Y	N		Y	N	Y	N	N	N		Y	N	N	N	Y	Y	Y	N	N	N			N	N			...	
dyn≤20%																N	N	N	N				N	N						N	...	
dyn≤40%										Y	N	N				N	N	N	Y	N	N	Y	N	N	Y	N	N	N	Y	Y	...	
dyn≤70%																Y	N	N													...	
Rs≤30%											Y	N	N				Y	N		Y	N		Y	N			Y	N	Y	N	...	
Ps≤50%																													Y	N	...	
Su≤50%																													Y	Y	...	
Choices																															...	
EXCELL	x	x		x	x			x							x																...	
Z-hash			x				x							x										x				x			...	
Quantile						x			x	x						x				x	x		x		x					x	...	
PLOT											x						x		x			x				x			x		...	
BANG												x	x					x									x				...	

Table 1. Matching the application properties to implementation algorithms

For **conditions**, "Y" indicates a satisfied condition; "N" represents an unsatisfied condition and " " states unconcerned condition.
For **choices**, "X" indicates a selected algorithm.

Construction of h(x) on heuristics

Available algorithm set

ALT = { ALT[i] for i = 1, 2, ..., x}

Property set

PRT = { Pj for j = 1, 2, ..., y }
= {d(even), Rs, Dyn, Ps, Ds}

Heuristic function

h(x) = j where max(hi(x) for i = 1, 2, ..., y) = hj(x)

if there are more than one hi(x) which have the same value and happen to be the maximum then we can choose the simplest one among them.

$$hi(x) = \sum_{Pj=1}^{Pj=x} h_{Pj(ALT[i])}$$

for i = 1, 2, ..., y

$$h_{Pj(i)} = \begin{cases} 0 & \text{if Pj is irrelevant to ALT[i]} \\ w(j) & \text{if Pj has j degree significance to ALT[i]} \end{cases}$$

Decision rules on h_{Pj}(ALT[i])

Deven = { d1, d2, d3, d4 }
= { 20%, 25%, 40%, 50% }

DY = { dy1, dy2, dy3, dy4, dy5 }
= { 10%, 20%, 30%, 40%, 50% }

Ps = { p1, p2 p3, p4, p5}
= { 10%, 20%, 30%, 40%, 50%)

Rs = { r1, r2, r3, r4, r5 }
= { 10%, 20%, 30%, 40%, 50% }

(1) The degree of even distribution

Deven = { d1, d2, d3, d4 }
= { 20%, 25%, 40%, 50% } is equivalent to

d(even) = { ≤ d1, ≤ d2, ≤ d3, ≤ d4, > d4 }
= { ≤ 20%, ≤ 25%, ≤ 40%, ≤ 50%, > 50% }

$h_{Deven(EXCELL)}$ = { 4, 5, 3, 2, 1 }
$h_{Deven(z\text{-}hashing)}$ = { 5, 4, 3, 2, 1 }
$h_{Deven(quantile\text{-}hashing)}$ = { 3, 4, 5, 2, 1 }
$h_{Deven(PLOP\text{-}hashing)}$ = { 3, 4, 5, 2, 1 }
= $h_{Deven(quantile\text{-}hashing)}$
$h_{Deven(BANG)}$ = { 1, 2, 3, 4, 5 }
$h_{Deven(R\text{-}tree)}$ = { 0 } irrelevant

Interpretation

d(even) = { ≤ 20%, ≤ 25%, ≤ 40%, ≤ 50%, > 50% }

$h_{Deven(EXCELL)}$ = { 4, 5, 3, 2, 1 }

IF: d(even) ≤ 20%
THEN: h = 4
REASON: if the data are evenly distributed then the z-hashing is the best choice that leaves the EXCELL as the second choice.

IF: d(even) ≤ 25%
THEN: h = 5
REASON: if the data set is more or less evenly distributed then the EXCELL performs the best.

IF: d(even) ≤ 40%
THEN: h = 3
REASON: if the data set is less evenly distributed then the Quantile- and PLOP-hashing can deal with it better than the EXCELL.

IF: d(even) ≤ 50%
THEN: h = 2
REASON: if the data set is not evenly distributed then the Quantile- and PLOP-hashing and BANG file can deal with it better than the EXCELL.

IF: d(even) > 50%
THEN: h = 1
REASON: if the data set is non-uniformly distributed then the EXCELL can not deal with it well.

The value is assigned by a relative comparison with other algorithms, i.e. a set of properties of a feature can be selected to reflect a spectrum, a particular algorithm should have an optimal position along this spectrum, which is determined by experience and performance evaluation. Similarly for other parameters.

The values in h_{Pj}(ALT[i]) are given based on the judgement of how an algorithm is capable of dealing with a property and they are assigned corresponding to the order in the property list. For property d(even) there is a set of upper and lower boundaries given in the property list. If an algorithm, say the EXCELL, deals with non-uniform distribution **reasonably** well then it means that the EXCELL handles uneven data distribution better than some algorithms, but some other algorithms cope with uneven data distribution better than EXCELL. By referencing the ability of other algorithms we give the highest value to h_{Pj}(ALT[i]) for the case of d(even) < 40%. If x, y represents other algorithms and x does not handle uneven data as well as EXCELL whereas y can handle it better, then hd(even)(x) < $hd_{(even)}$(EXCELL) < $hd_{(even)}$(y). So the h(x) value is determined by evaluating the most suitable range of a feature that an algorithm can perform. For instance, if algorithm A handles unevenly distributed data better than that of algorithm B, algorithm A gain a higher score for unevenly distributed data than that of algorithm B. Here we assume that the higher the score, the more suitable is an algorithm.

(2) Range search rate Rs

Rs = { r1, r2, r3, r4, r5 }
= { 10%, 20%, 30%, 40%, 50% } is equivalent to
r(s) = { ≤ r1, ≤ r2, ≤ r3, > r5, > r4 }

h_{Rs}(EXCELL) ={ 3, 4, 5, 1, 2 }
h_{Rs}(z-hashing) = { 1, 2, 3, 5, 4 }

h_{Rs}(quantile-hashing) = h_{Rs}(z-hashing)
h_{Rs}(PLOP-hashing) = { 3, 5, 4, 1, 2 }
h_{Rs}(BANG-file) = hRs(z-hashing)
h_{Rs}(R-tree) = { 0 } irrelevant

(3) Insertion or deletion rate
Dyn = Ir + Dr
DY = { dy1, dy2, dy3, dy4, dy5 }
= { 10%, 20%, 30%, 40% 50% }
Dyn = { ≤ dy1, ≤ dy2, ≤ dy3, ≤ dy4, > dy5 }

h_{Dyn}(EXCELL) = { 4, 3, 5, 2, 1 }
h_{Dyn}(z-hashing) = { 5, 4, 3, 2, 1 }
h_{Dyn}(quantile-hashing) = h_{Dyn}(z-hashing)
h_{Dyn}(PLOP-hashing) = { 1, 2, 3, 4, 5 }
h_{Dyn}(BANG file) = { 1, 2, 3, 5, 4}
h_{Dyn}(R-tree) = { 0 } irrelevant

(4) Point search rate
Ps = {p1, p2, p3, p4, p5}
= {10%, 20%, 30%, 40%, 50%} is equivalent to
p(s) = {≤ p1, ≤ p2, ≤ p3, > p4, > p5}
= {≤ 10%, ≤ 20%, ≤ 30%, > 40%, > 50%}

h_{Ps}(EXCELL) = { 5, 4, 3, 2, 1 }
h_{Ps}(z-hashing) = { 1, 2, 3, 4, 5 }
h_{Ps}(quantile-hashing) = { 1, 2, 3, 5, 4}
h_{Ps}(PLOP-hashing) = { 1, 2, 5, 3, 4 }
h_{Ps}(BANG-file) = { 4, 5, 2, 3, 1 }
h_{Ps}(R-tree) = { 0 } irrelevant

(5) Data set size
|Ds| = { 1, 2, 3, 4, 5 }
|Ds| is defined by five degrees from small to large because the data set size is a machine dependent factor (a data set which is considered to be large in a PC environment may not be considered to be large in a mainframe environment) so that we leave the decision until the system is being installed.

$h_{|Ds|}$(EXCELL) = { 5, 4, 3, 2, 1 }
$h_{|Ds|}$(z-hashing) = { 1, 2, 4, 5, 3 }
$h_{|Ds|}$(quantile-hashing) = { 1, 2, 3, 5, 4}

$h_{|Ds|(PLOP\text{-}hashing)}$ = h|Ds|(quantile-hashing)
$h_{|Ds|(BANG\text{-}file)}$ = { 1, 2, 3, 4, 5 }
$h_{|Ds|(R\text{-}tree)}$ = { 0 } irrelevant

(6) Differences among local data densities
To obtain a measure of the difference we evaluate the following:

$$Ldx1 = \sum_{y=1}^{y=sy} C[1, y]$$

$$Ldx2 = \sum_{y=1}^{y=sy} C[2, y]$$

.
.
.

$$Ldxsx = \sum_{y=1}^{y=sy} C[sx, y]$$

$$Ldy1 = \sum_{x=1}^{x=sx} C[x, 1]$$

$$Ldy2 = \sum_{x=1}^{x=sx} C[x, 2]$$

.
.
.

$$Ldysy = \sum_{x=1}^{x=sx} C[x, sy]$$

Lx = { Ldxi for i = 1, 2, ..., sx }
Ly = { Ldyi for i = 1, 2, ..., sy }
Lx' = { Ldx'i for i = 1, 2, ..., sx }
Ly' = { Ldy'i for i = 1, 2, ..., sy }

Lx' and Ly' are two sets that are arranged in an ascending order. The degree of local density differences can be calculated by Dxy as follows.

$$Dxy = \frac{\sum_{i=1}^{i=sx-1} Ldx'i + \sum_{i=1}^{i=sy-1} Ldy'i}{(sx + sy) \times b}$$

Here sx and sy are the number of slices on the dimension x, y respectively.

Dxy = {dx1, dx2, dx3, dx4, dx5 }
= { 10%, 20%, 30%, 40%, 50% }

d(xy) = { ≤ dx1, ≤ dx2, ≤ dx3, ≤ dx4 ≤ dx5, > dx5 }

^{h}Dxy(EXCELL) = { 5, 6, 4, 3, 2, 1 }
^{h}Dxy(z-hashing) = { 6, 5, 4, 3, 2, 1 }
^{h}Dxy(quantile-hashing) = { 1, 2, 3, 4, 5, 6 }
^{h}Dxy(PLOP-hashing) = hDxy(quantile-hashing)
^{h}Dxy(BANG-file) = hDxy(z-hashing)
^{h}Dxy(R-tree) = hDxy(quantile-hashing)

(6) If the database is a non-zero sized object one ALT[6] may be chosen. The R-tree algorithm is especially suitable for dynamic object data because the size of a region can be changed accordingly, without difficulty.

To make a decision the heuristic function h(x) used is based on these parameters obtained from a given application. Different weights can be assigned to different parameters according to requirements and the algorithm concerned. For each parameter value a h(x) is calculated. For example, if an application has properties of P1: 20% < d(even) < 25%, P2: 20% < Rs < 30%, P3: 30% < Dyn < 40%, Ps not concerned,P4: |Ds| = 3, P5: Dxy < 40 then we get:

	EXCELL	z-hashing	quantile -hashing	PLOP -hashing	BANG
^{h}P1(x)	5	4	4	4	2
^{h}P2(x)	5	3	3	4	4
^{h}P3(x)	2	2	2	4	5
^{h}P4(x)	3	4	3	3	3
^{h}P5(x)	3	3	4	4	3
hi(x)	18	16	16	19	17

thus h(x) = max(hi(x))

= PLOP-hashing for $h_{PLOP\text{-}hashing(x)}$

= 19.

There may be cases where more than two algorithms bear the same value of the heuristic function hi(x) so that either the algorithm with the least complexity will be chosen or a choice is made by the users.

ALT[1]: Cannot deal with range search well,
cannot guarantee one secondary storage access for very large data set.

ALT[2]: Cannot deal with dynamic situation well,
cannot deal with unevenly distributed data well,
implementation is more costly than ALT[1].

ALT[3]: Cannot deal with dynamic situation well,
implementation is more costly than ALT[2].

ALT[4]: Cannot deal with range search well,
implementation is more costly than ALT[3].

ALT[5]: Implementation is more costly than ALT[4].

Explanation of the heuristics

For each property of an application we have given a range of values. They are used to determine the category of an application for that property. We have also listed the corresponding values (created by heuristics) given to to each of these ranges for each algorithm. These values indicate the ability of an algorithm to deal with the concerned property. They are assigned on the basis of heuristic judgement by comparing different algorithms. The explanation is derived from the interpretation of the property list and the capability of each algorithm. For each property list and heuristic list a fuzzy set can be established to represent the corresponding meaning. The values given in the list are by no means accurate and the resolution may not be high enough. The high resolution means that the number of values in a list increases. They are given to form a system framework. Further tuning can be introduced by monitoring the system performance and observing the critical

changes. Tuning will be discussed in section 4.5.

An example

d(even) = { ≤ 20%, ≤ 25%, ≤ 40%, ≤ 50%, > 50% }

ex(even distribution) = { highly, more or less, average, less, un- }

To explain an application, say, d(even) = 20%, the system will generate: " highly even distribution ".

Since we create heuristic values by assigning the highest to represent high capability, the explanation corresponds to the position and the value assigned.

$ex(hp_{j(x)})$ = { ≥ 5 --> very well,
= 4 --> well,
= 3 --> not so well,
= 2 --> not well,
= 1 --> badly,
= 0 --> irrelevant }

The list hd(even)(EXCELL) = { 4, 5, 3, 2, 1 } will be explained by referencing ex(even distribution) as:
The EXCELL algorithm deals reasonably with highly evenly distributed data; very well with more or less even data distribution; not so well with average evenly distributed data; not well with less evenly distributed data; and badly unevenly distributed data.

We have to be aware that the heuristics are applied by a comparison of different algorithms. The explanation " to deal with highly even distribution well, but not very well" seems nonsense otherwise, for we know that evenly distributed data can always be dealt with well. The reason that the explanation says "well" is that the z-hashing can deal with evenly distributed data better than the EXCELL.

4.5. Dynamic monitoring and tuning of a physical database

As we stated in chapter 3 the system should be able to monitor the performance of a database in order to satisfy users' requirements. The users' requirements are obtained from USI and stored as part of DBP for an application. During the running of a database a chosen algorithm may not satisfy the requirements. The reason is twofold: (a) the incomplete information supplied by the system or user is inaccurate so a wrong AAP algorithm has been chosen to process the data set; (b) the dynamic changes to the data set have changed the properties of the application. To deal with both situations, the application properties need to be reevaluated. Depending on the system implementation, either the data about the previous estimation is printed, or kept in the system to compare with the current database status. If a different AAP can be found with required similarity then the data set is reorganised accordingly, otherwise the system will display how an algorithm is chosen for the application. The reasoning process will be discussed in the following section.

To monitor the performance the system will use the database profile and consult a set of rules to decide on what to do. The data required for a decision are derived from the information stored for the database profile about the concerned data set. These data are transformed into the same format as AAP in order to carry out the similarity comparison. These data include data space resolution r, even data degree d(even), dynamic factor Dyn, range search rate Rs, number of empty grid cells Nempty, number of overflow grid cells Nover, local data density Ld, and query frequencies F. They are calculated from the current database profile and then used for similarity comparison and selection of an implementation algorithm. If needed the data can also be used to compare with the original database profile.

Dynamic tuning for the database

There are two aspects to be considered in the tuning process. One is to apply different control functions and heuristics to partition the data space and to optimise the advantages of a chosen algorithm, i.e. tuning by improving an individual algorithm; the other is to reorganise the data set by a new implementation algorithm, that is, tuning by changing the implementation.

4.5.1. Tuning by improving the individual algorithm

The purpose of tuning an individual algorithm is to increase storage utilisation, to reduce processing time and to relax constraints imposed on the system. For

example, a hashing algorithm may require contiguous storage space.

With the EXCELL algorithm the problem is likely to arise that a split may make the condition |idx| < M invalid and therefore, one more secondary storage access is required for a retrieval. Consequently it may be desirable to change the implementation if the access speed is critical.

With the z-hashing algorithm several tuning strategies may be introduced. These strategies improve the performance as a whole to maintain the fast access speed and maximise the storage utilisation.

Tuning an individual algorithm is done by a set of control functions and heuristics, geared to different applications. To perform the task, the application is analysed and rules are derived for the above mentioned goals.

Tuning the z-hashing algorithm by overflow handling

Since uneven data distribution may introduce data holes tuning can be done by making use of the space reserved for these holes. When there are a number of data holes for a data set the corresponding data buckets can be used to hold overflow data. Thus the storage utilisation can be improved. In addition, a choice can be made as to whether overflow handling is necessary or not. When there are few overflow grid cells, removing overflow by splitting will be more economic than overflow handling; otherwise separate overflow handling may be applied to gain better performance.

Using buckets for empty cells to hold an overflow record.
OVR1 = { Nempty/B ≥ 10% --> overflow tuning}.

To implement such an idea the following data structure is designed.

z-code	status
•	
•	
•	

where a z-code gives an empty cell address and the status indicates whether it is used or not. An insertion to the table is made when an empty grid cell occurs; a deletion from the table is made when an empty cell becomes non-empty.

The rule for the z-hashing overflow processing
OVR2 = { Nover/B > 20% --> overflow handling }
OVR3 = { d(even) < 10% --> without overflow handling }

explanation
The rule OVR2 determines when the empty buckets are to be utilised for overflow. When the premise condition is satisfied the overflow tuning will be invoked. The tuning will create necessary data structures and extract required information to improve the storage utilisation for the z-hashing.

The rule OVR3 determines when the overflow handling is necessary. For a fairly evenly distributed application the number of overflow grid cells is small, so that the splitting can be minimal, to remove these overflow grid cells.

Improving the z-hashing algorithm by data space segmentation
As analysed before the z-hashing algorithm cannot cope with uneven data distribution well and has less flexibility to deal with a dynamic situation. However, it does offer a good performance in terms of access time for both point and range

searches. To gain its advantage of fast access time the entire data space may be divided into a number of smaller subspaces, each subspace may apply different resolutions. Such an improvement will reduce the number of empty holes and make the data rearrangement localised; but this improvement can only be made for certain data distributions. The ideal situation is that each subspace has a different data density so that it naturally fits the purpose of dividing the entire data space into several subspaces. The implementation idea is illustrated in Appendix A7.

In this section we turn our attention to how these rules will be derived to support this idea. To analyse the data distribution different levels of resolutions are used to calculate C[i] for i = 1, 2, ..., ri. To measure the data distribution for this purpose we introduce resolution difference RD.

$$RD = \left\lceil \frac{\sum_{i=1}^{i=ri} C[i]}{\min_{j=1}^{j=ri} (C[j] \mid C[j] \geq b)} \right\rceil$$

An example

ri = 4: b = 10, C[1] = 40, C[2] = 40, C[3] = 80, C[4] = 320.

$\min_{j=1}^{j=ri} (C[j] \mid C[j] \geq b) = 40$

RD = (40/40 + 40/40 + 80/40 + 320/40) = 12

where b is the expected bucket size for the data set, i.e. if b= 512 bytes and the record length R = 50 then the expected b will be 10.

Rules for the z-hashing data space segmentation

ZR1 = { (ri = 4: RD ≥ 4 + 2 = 6) --> strategy1 }

ZR2 = { (ri = 8: RD ≥ 8 + 4 = 12) --> strategy2 }
ZR3 = { (ri = 16: RD ≥ 16 + 8 = 24) --> strategy3 }
ZR4 = { (ri = 32: RD ≥ 32 + 16 = 48) --> strategy4 }
ZR5 = { (ri = 64: RD ≥ 64 + 32 = 96) --> strategy5 }

Explanation

EZR1

When the resolution is 4 we calculate the resolution differences RD. The lower bound is to use a four entry index file is RD ≥ 4 + 2 = 6, the first item 4 catering for the balance of introducing an index and the second item 2 is an average number of grid cells which are likely to have the same minimum data items in them.

EZR2

When the resolution is 8 we calculate the resolution differences RD. The lower bound is to use a eight entry index file is RD ≥ 8 + 4 = 12, the first item 8 catering for the balance of introducing an index and the second item 4 is an average number of grid cells which are likely to have the same minimum data items in them.

Similarly we have the same explanation text for the other rules. To store these explanations efficiently we can store the text once with the changed part supplied, i.e. the following data are kept:

xi	yi	zi	for i = 1, 2, ..., 5
EZR1	4	2	6
EZR2	8	4	12
EZR3	16	8	24
EZR4	32	16	48
EZR5	64	32	96

EZRi when the resolution is ri we calculate the resolution differences RD. Further reduction of storage can be introduced by merging common parts of each rule. The lower bound to use a eight entry index file is RD ≥ xi + yi = zi, the first item xi catering for the balance of introducing an index and the second item yi is an average number of grid cells which are likely to have the same minimum data items in them.

Improving the Quantile-hashing and the PLOP-hashing by partitioning control

As illustrated before both quantile- and PLOP-hashing allow partitioning control. If one of these two algorithms is chosen, the tuning process will mainly depend on selecting the dividing position when a split is triggered. These controls can be divided into several categories and used according to the properties of an application.

Rules for partition control

$$PCR1 = \{ \max_{sij=1}^{sij=simax} (Ld(sij) = sa) \text{ --> split on slice a in the } i^{th} \text{ dimension) } \}$$

$$PCR2 = \{((\max_{ki=k1}^{ki=km} (f(ki) = kb) \;\&\; (sbmax \leq \frac{f(kb)}{\min_{ki=k1}^{ki=km} (f(ki))=kc} \times scmax)) \text{ > split in } b^{th} \text{ dimension}\}$$

$$PCR3 = \{ \text{ split sij into sij(1) and sij(2) <=> } |N(sij(1))| \approx |N(sij(2))| \ \}$$

Tuning the R-tree algorithm by representation

A useful property of the R-tree is that its region boundaries are not determined in advance. Thus dynamic changes can be easily made. However, the flexibility is obtained by recording boundary ranges for each dimension of a region. A region identifier is represented as I = {(li, ui) for i = 1, ..., m, where li is the lower bound in the i^{th} dimension and ui is the upper one. The representation may become a problem for the R-tree method because it may occupy too much space which means increasing the index height and consequently increases the search time, especially when m is large.

To tune the R-tree method several alternative representations are introduced.

Representing a region by a starting point and an offset

representation

obj = { (si, li) for i = 1, ..., m }

where: si -- starting position

li -- offset from the starting position

This representation is suitable where the boundary si needs more space than the offset li. The maximum value of the boundary can be estimated according to the range of space; and the maximum value of an offset can be estimated by the largest object or region. When |si| > |li| the representation can be applied. The gain can be measured by $\sum_{i=1}^{n} (|si| - |li|)$, where n is the number of regions.

Representing a region by relative position

representation

SP = {sp1, sp2, ..., spm}
where SP is a set of starting positions of the data space.

LP = {lp1, lp2, ..., lpm}
where LP is a set of offsets relative to the starting position.

A region is represented as:
obj = { (lpi1, lpi2) for i = 1, 2, ..., m }

This presentation is similar to the coordinate transformation used in algebra by moving a data space to the coordinate origin. The method is suitable if the starting value is large and n is significant. The gain can be measured by

$$2 \times \sum_{i=1}^{i=n} (|spi| - |lpi|) - \sum_{j=1}^{j=m} |spj| = 2 \times \sum_{i=1}^{i=n} (|spi| - |lpi|) - m \times |sp1|$$

where n is the number of grid cells and m is the dimensionality.

As the knowledge of each algorithm develops new tuning rules can be added to the system for the best use of each algorithm.

4.5.2. Tuning by changing implementation

When little improvement can be gained by tuning the individual algorithm, a new algorithm may be required to replace the current one. The database profile is examined and the rules concerning selection of algorithms are inspected to arrive at a new solution. This process is carried out by eliminating rules, similarity comparison or the heuristic functions. After a new algorithm is selected the relevant parts of the database profile are updated.

4.6. Adding new knowledge to the system

When a new class of an application is added to the system new knowledge needs to be added. Adding new knowledge to the system is performed by the following rules.

Recording the similarity

RSR1 = { 50% < s(x) ≤ 60% <--> idk(ATj) for k = 1, 2, ..., y1
j = 1, 2, ..., x1 }
RSR2 = { 60% < s(x) ≤ 70% <--> idk(ATj) for k = 1, 2, ..., y2
j = 1, 2, ..., x2 }
RSR3 = { 70% < s(x) ≤ 80% <--> idk(ATj) for k = 1, 2, ..., y3
j = 1, 2, ..., x3 }
RSR4 = { 80% < s(x) ≤ 90% <--> idk(ATj) for k = 1, 2, ..., y4
j = 1, 2, ..., x4 }
RSR5 = { 90% < s(x)≤ 100% <--> idk(ATj) for k = 1, 2, ..., y5
j = 1, 2, ..., x5 }

The set of rules tells us the similarity degree of existing data set idk compared with the application type ATj for the database, i.e. if a data set idk is within the range of similarity used in the premises (LHS) then it is recorded as the conclusion part in these rules or vice versa. It implies information (idk, ATx, s(x)). Similarly the following rules will record the heuristics used and the performance evaluated for these data sets.

Recording the heuristics

HBR = { h1, h2, h3, h4, h5 } = { 5, 10, 15, 20, 25 }

HBR records the heuristic function boundaries, since the boundary may change it is beneficial to record them separately for easy maintainability.

for applications

HR1 = { h1 < h(x) ≤ h2 <--> idk(ALT[j]) for k = 1, 2, ..., y1
j = 1, 2, ..., x1}
HR2 = { h2 < h(x) ≤ h3 <--> idk(ALT[j]) for k = 1, 2, ..., y2
j = 1, 2, ..., x2 }
HR3 = { h3 < h(x) ≤ h4 <--> idk(ALT[j]) for k = 1, 2, ..., y3

j = 1, 2, ..., x3 }

HR4 = { h4 < h(x) ≤ h5 <--> idk(ALT[j]) for k = 1, 2, ..., y4
j = 1, 2, ..., x4 }

for system

HR5 = { h1 < h(x) ≤ h2 <--> ATk(ALT[j]) for k = 1, 2, ..., y1
j = 1, 2, ..., x1}

HR6 = { h2 < h(x) ≤ h3 <--> ATk(ALT[j]) for k = 1, 2, ..., y2
j = 1, 2, ..., x2 }

HR7 = { h3 < h(x) ≤ h4 <--> ATk(ALT[j]) for k = 1, 2, ..., y3
j = 1, 2, ..., x3 }

HR8 = { h4 < h(x) ≤ h5 <--> ATk(ALT[j]) for k = 1, 2, ..., y4
j = 1, 2, ..., x4 }

The set of rules stores a value for heuristic function h(x), for a data set idk or for an ATx that employs algorithm ALT(j), i.e. {idk, ALT[k], h(x)} and {ATx, ALT[k], h(x)}.

Recording the performance

(a) access speed

for applications

PR1 = { Tsec <--> idk(ALT[j]) for k = 1, 2, ..., y1 j = 1, 2, ..., x1 }

PR2 = { 2 x Tsec <--> idk(ALT[j]) for k = 1, 2, ..., y2 j = 1, 2, ..., x2 }

PR3 = { 3 x Tsec <--> idk(ALT[j]) for k = 1, 2, ..., y3 j = 1, 2, ..., x3 }

PR4 = { 4 x Tsec <--> idk(ALT[j]) for k = 1, 2, ..., y4 j = 1, 2, ..., x4 }

PR5 = {(T > 4 xTsec) <--> idk(ALT[j]) for k = 1, 2, ..., y5 j = 1, 2, ..., x5 }

for the system

PR6 = { Tsec <--> ATk(ALT[j]) for k = 1, 2, ..., y1, j = 1, 2, ..., x1 }

PR7 = { 2 x Tsec <--> ATk(ALT[j]) for k = 1, 2, ..., y2 j = 1, 2, ..., x2 }

PR8 = { 3 x Tsec <--> ATk(ALT[j]) for k = 1, 2, ..., y3 j = 1, 2, ..., x3 }

PR9 = { 4 x Tsec <--> ATk(ALT[j]) for k = 1, 2, ..., y4 j = 1, 2, ..., x4 }

PRa = {(T > 4 xTsec) <--> ATk(ALT[j]) for k = 1, 2, ..., y5 j = 1, 2, ..., x5 }

(b) storage utilisation

for the application

UR1 = { 50% < Su < 60% <--> idk(ALT[j]) for k = 1, 2, ..., y3 j = 1, 2, ..., x3 }
UR2 = { 60% < Su < 70% <--> idk(ALT[j]) for k = 1, 2, ..., y3 j = 1, 2, ..., x3 }
UR3 = { 70% < Su < 80% <--> idk(ALT[j]) for k = 1, 2, ..., y3 j = 1, 2, ..., x3 }
UR4 = { 80% < Su < 90% <--> idk(ALT[j]) for k = 1, 2, ..., y3 j = 1, 2, ..., x3 }
UR5 = { 90% < Su < 100% <--> idk(ALT[j]) for k = 1, 2, ..., y3 j = 1, 2, ..., x3}

for the system

UR6 = { 50% < Su < 60% <--> ATk(ALT[j]) for k = 1, 2, ..., y3
j = 1, 2, ..., x3 }

UR7 = { 60% < Su < 70% <--> ATk(ALT[j]) for k = 1, 2, ..., y3
j = 1, 2, ..., x3 }

UR8 = { 70% < Su < 80% <--> ATk(ALT[j]) for k = 1, 2, ..., y3
j = 1, 2, ..., x3 }

UR9 = { 80% < Su < 90% <--> ATk(ALT[j]) for k = 1, 2, ..., y3
j = 1, 2, ..., x3 }

URa = { 90% < Su < 100% <--> ATk(ALT[j]) for k = 1, 2, ..., y3
j = 1, 2, ..., x3 }

Rule for relating the AT and the ALT

RLRi = { ALT[i] <--> AT1, AT2, ..., ATxi }

for i = 1, 2, ..., z where z is the number of algorithms.

For this set of rules they record all application types that use ALT[i] as the implementation algorithm.

These above rules record information which can be summarised as:

Similarity

(idk, ATx, s(x))

heuristics

{idk, ALT[k], h(x)}

{ATx, ALT[k], h(x)}

access speed

{idk, ALT[k], T}

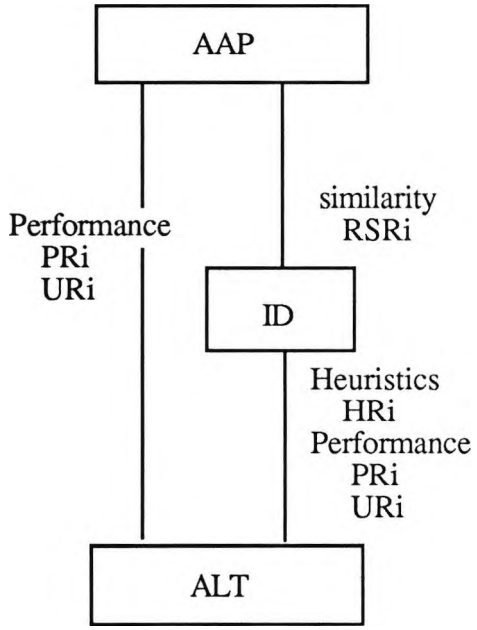
{ATx, ALT[k], T}

Storage utilisation

{idk, ALT[k], Su}

{ATx, ALT[k], Su}

These rules establish relationship between an AAP(ATi) and a data set (ID) for an application, between an AAP and an ALT, as well as between ID and ALT. It is shown briefly as:



This relationship builds up a reasoning path. If an instance in one set is known then the related instances in other sets can be derived. We now describe how to use the information to refine the system.

Inspecting performance

IRi = { if Su(idi) within the range of URi premise then get all action / conclusions (RHS) from URk for k = 6, 7, ..., i }

This set of rules tell us that for the expected storage utilisation all satisfied data sets are extracted from rule URik for k = 6, ..., a. If the data set idi is not in the extracted set IRi then idi is of lower storage utilisation than that expected. The other URk for k = i + 1, ..., a can be examined to see which URi idi belongs to in order to work out the differences between the expected storage utilisation and the actual one. Similarly for an ATi in the AAP.

IRj = { if Nsec within the range of PRj premise then get all conclusions from PRk for k = 1, 2, ..., j }.

This set of rules tell us that for the expected number of secondary storage accesses all satisfied data sets are from rule PRik for k = 1, 2, ..., 5. If the data set idi is not in this extracted set IRi then idi is of slower access speed than that expected. The other PRk for k = 6, ..., a can be examined to see which PRi idi belongs to so to work out the differences between the expected access speed and the actual one. Similarly for ATi in the AAP.

Adding new information

Using heuristics and experience involves tuning. Heuristic judgement may not necessarily be correct all the time and therefore, the system should provide facilities to refine rules used for judgement to enable new knowledge to be added.

As the performance, range of heuristics, and similarities have been recorded in the system for a specific application the probability of success can be derived from the recorded information. For instance, if an application requires 70% storage utilisation (Su = 70%) and one secondary storage access (T = Tsec) then after selecting an algorithm for the data set idz the relevant PRi, URi and HRi rules are checked. If an algorithm is selected by the similarity comparison and ATx is the chosen image then PRi, URi and RSRi rules will be examined; if an algorithm is selected by heuristics then PRi, URi and HRi rules will be inspected. For the former situation the performance of ATx can be obtained for the application with the same range of similarity.

The reasoning process is:

(a) s(x) --> idk(ATz) for k = 1, ..., a; z = 1, ..., b (RSRi rule)
(b) ATx --> ALT[j] (RLRi rule)
(c) idk(ALT[j]) --> Su (URi rules)
(d) idk(ALT(j)) --> T (PRi rules)

From the reasoning processes a set of data can be derived.
{s(x), ATx, Su, T} --> pp1
where s(idz(ATx)) is the value of the similarity degree for the current application related to ATx; ATx is the application type chosen for deriving the solution for idz; Su and T are the performance values; pp1 is the percentage of all applications, which achieve a performance at least as good as Su or T, or (Su and T) against the total number of applications. They are based on the solution by similarity comparison with ATx (pp1(T), pp1(Su), pp1(T, Su)). Furthermore, pp1 gives the degree of confidence for using ATx as an application type to derive a solution. When the value of pp1 is low the relevant parts of the rule base will be examined; whereas when the pp1 is high the value of the similarity degree can be used as an upper threshold to speed up the rule search. Similarly for the algorithm chosen by heuristics we can derive:
{idz, h(x), ALT[j], Su, T} --> pp2. An illustration of relationship between applications and various indices (similarity, speed, storage utilisation) is shown in Figure 4.35.

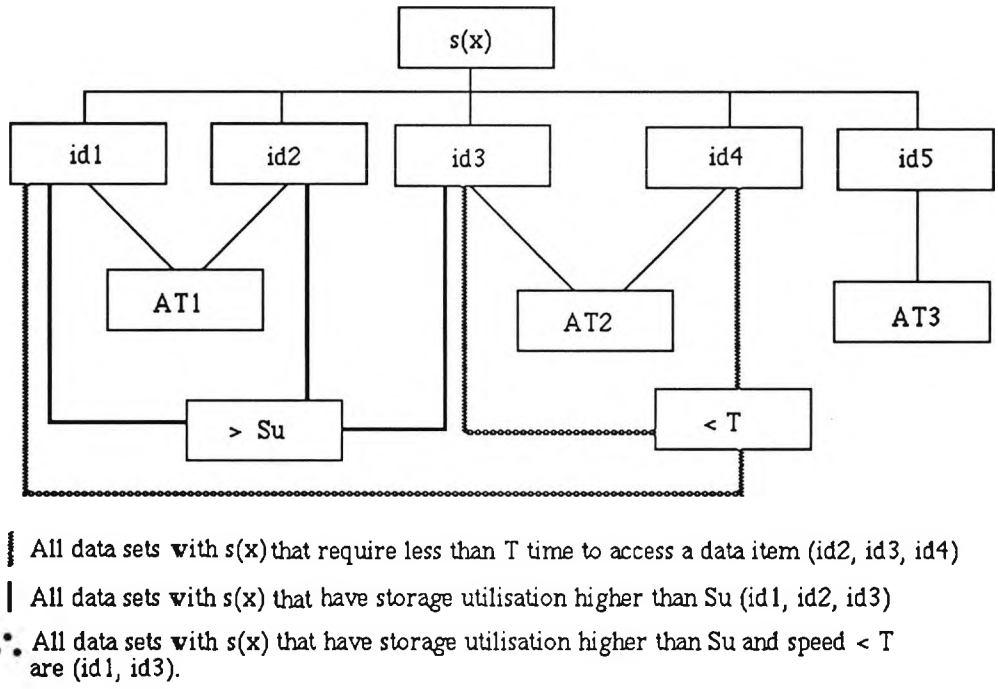


Figure 4.35. Relationship between applications and relevant indices.

In the diagram, we have:

s(x) - a particular similarity degree range

idi - all data sets with s(x) as their similarity degree range. There are five data set in this example (i = 1, 2, 3, 4, 5)

Su - a particular storage utilisation threshold

T - a particular access time threshold

Refining rules

Having derived pp1 and pp2 an analysis can be carried out when pp1 or pp2 is low. The low value indicates where refinement may be desired.

When selecting an algorithm by similarity comparison the upper bound needs refining.

When selecting an algorithm by a heuristic function upper or lower bounds used in the heuristic rules need refining. This refinement is done by evaluating performance of the algorithm through sensitivity analysis of major parameters.

Furthermore, the AAP with different application types may have similar properties. The system can classify them as the same application type if the similarity degree of various application features is within the acceptable range among ATi for i = 1, ..., x; even if they appear to be different application classes.

4.7. Reasoning process

After a decision has been made the user may want to know the reason as to why the implementation algorithm was chosen. In response the system should be able to display the reasoning procedure for an application. The reasoning process is supported by storing relevant information during the similarity comparison about properties of the chosen AAP. In the system this function is carried out by the following rules.

The reasoning rules

RRj = { ALT[j] <--> id1, id2, ..., idyi }
for j = 1, 2, ..., z where z is the number of algorithms

For this set of rules they record all data set in a database profile that use ALT[i] as the implementation algorithm.

STR = { ((Su(idi) | Nsec(idi)) = unsatisfactory) -->
(∃ idi(ATi) --> similarity range) |
(∃ idi(ALT[j]) --> heuristic value) |
(∃ idi --> ALT[j] }

If performance status in terms of storage utilisation and access time for a data set idi is unsatisfactory then rules numbered as RSRi, HRi and RRj are inspected.

4.8. A complete example

Here follows an example which explains how the problem of a given data set using m-d algorithm is solved by the system. In this example, we first illustrate the steps taken to arrive at the solution, then we illustrate by means of a diagram the decision making sequence.

Initial algorithm selection

The USI acquires information to construct an initial profile for a data set. The information contains the following aspects.

Initial algorithm selection

ACi	: an application class
data set	: Ds = { d1, d2, ..., dn }
search space	: di = (ai1, ai2) for i = 1, 2, ..., n
average length of the data items	: R
the access mode	: random access
data set life span	: tm
bucket size	: b
packing density	: p > 75%
the domain of the data space	: D = D1 X D2
estimated average data item size	: Re
range search rate	: Rs
available memory size	: M (estimated)

The derived parameters are

number of buckets required	: Nreq
number of buckets to be used (estimation)	: Ntot
data distribution	: Nover, Nempty

If any information is missing from a user then the USI will assign the data set with a status of "information missing". Within the system there are a set of application classes and under each class there will be a number of application types. As defined before the application class is: ACi = { application area, definition }. For instance, a theatre booking or a film booking application, and an airline booking application may be classified as class AC1 = { ticket booking, (booking, cancelling, reserving, scheduling) } and with different types AT11, AT12, AT13. A new application,

say, an airline booking system, will be classified as class AC1, containing AT3. Classifying an application is done by selecting the definition and the functions shown on the screen. The screen may first show the high level as:

Screen 1

Application areas

Selecting the correspond application class

AC1.	ticket booking
AC2.	employee/student management
AC3. CAD:	computer aided design
AC4. CAI:	computer aided instruction
AC5.	traffic control system

.

.

.

If AC1 is chosen then the next level is shown as:

Definitions

Selecting the relevant functions and data sets for your application

D1. customer	F1. booking
D2. seat	F2. cancelling
D3. flight/programme	F3. reserving
	F4. scheduling

Do you want to see the application types? Y/N

If the answer is Y then the following screen will be shown

Application types

Choose a correspond application type

AT11. theatre booking system

AT12. cinema booking system

AT13. airline booking system

The AT13 will be selected and all missing information will be gained from the stored AAP profile for AT13. However, if the new application is a student management application and the similar application class is AC2, then missing information may be obtained from AT21 (where AT21 is the only application type

which belongs to AC2, and AT21 is an employee management system). If, on the other hand, there is no application class matches to the new application, then the system should be able to generate some information for the missing bits. The system has to deal with three situations respectively:

(a) An application type is in the AAP.
As shown above when AT13 is selected we transform the new application to AAP scale and then work out the missing information;

(b) An application class is in the system but not the same type.
In such a situation the functions are ticked for the system to determine which ATij can be chosen to derive the missing information;

(c) There is no such application class in the system.
If there is no reference available in the system for missing information then the system will choose the implementation algorithm according to eliminating rules. When no eliminating rules are suitable the system has to make a choice on incomplete information. The differences from complete information will be shown in arriving at a conclusion. Incomplete information reduces the accuracy of a solution.

After gaining all the necessary information from the USI the following steps are performed:

(a) Examination of the eliminating rules.
There are six such rules in the system and they are examined one by one. If one of them derives a solution to the application then the rule search terminates and an algorithm is chosen. The rule number (ERi for $i \in \{1, 2, ..., 6\}$) is recorded as part of reasoning trace in a template. All information is recorded as assumptions from which the solution is derived. If none of the eliminating rules work then the initial selection rules are triggered.

(b) Inspection of the initial algorithm selection rules.
If a choice is made then the initial selection is completed. The rule number and chosen algorithm are stored as a reasoning path for the user to reference when needed. If no decision is made then lindexl, d(even), Dyn, Ps, Rs are calculated. As explained before, these parameters may be missing. When they are omitted the system will assign 0 to its corresponding heuristic function, indicating ignorance of the parameter concerned.

In our example let us assume that the data set has a long life span, the access time is

less than 2 x Tsec and { |index| ≥ M, d(even) < 20%, Dyn = ?, Rs = ? } then referring to ATRi rules we have:

h(ATR1) = 0 + 0 + 0 + 0 = 0
h(ATR2) = 1 + 1 + 0 + 0 = 2
h(ATR3) = 1 + 0 + 0 + 0 = 1
h(ATR4) = 1 + 0 + 0 + 0 = 1
h(ATR5) = 1 + 0 + 0 + 0 = 1
h(ATR6) = 1 + 0 + 0 + 0 = 1

The max(h(ATRi) for i = 1, 2, ..., 6) = h(ATR2) and therefore, the z-hashing algorithm is initially chosen. If the user asks why the z-hashing is chosen, the system will select the explanation from explanation text numbered as EX2.1 - EX6.1 and EX2.2.

the z-hashing algorithm is selected

x1 = 1

EX2.1 = EX3.1 = EX4.1 = EX5.1 = EX6.1

The index file estimated for the EXCELL algorithm will exceed the capacity of main memory so that an extra access is required for point search if the EXCELL algorithm is chosen.

x2 = 1

EX2.2

The data distribution is even. It implies that fewer data holes will be introduced by the z-hashing.

x3 = x4 = 0

Not irrelevant.

In making this selection data about insertion and deletion as well as range searching are ignored.

Dynamic tuning and monitoring of the performance.

During the running of the database a tuning process is triggered when a performance deterioration is detected. As soon as this happens the database profile is examined for the relevant data set. There are two types of tuning: one is to improve the current chosen algorithm - an individual algorithm tuning; the other is to re-examine the properties of the data set to see if a new algorithm needs to replace the current one - thus changing the implementation.

(a) Tuning the current algorithm

Let us assume that the z-hashing algorithm is chosen for the data set. When the performance becomes unsatisfactory the tuning process is initiated. For the z-hashing algorithm implementation C[i] for i = 1, 2, ..., r are stored in the system so that the number of data items for resolution level r = 8, r = 16 can be evaluated and the z-hashing data space segmentation rule ZRi for i = 1, 2, ..., 5 (see page 165) can be applied to see if any improvement can be made by introducing a small sized index file for segmentation. If so the measures are taken and the system will treat each subspace as an independent data set, which will apply different resolutions catering for the data distribution. By segmentation the need for continuous space is relaxed, and by applying a different resolution, the number of empty data holes will be reduced. Hence the performance will be improved.

(b) Changing the implementation

When the characteristics of the data set including both data and the operations over it have changed to gear towards an algorithm that is not in use, the performance will be improved by applying a new implementation algorithm. The database profile is re-examined and the similarity comparison or the heuristic rules are used to select a new solution.

When the algorithm is not an initially chosen algorithm but determined by the similarity rules then the degree of similarity and the ATi selected are recorded as the rule performance. It reflects the possibility of correctness in terms of similarity degree and the adopted algorithm. After storing rule performance data an AAP is selected on its class and types. Scale transformation is performed and parameters of DD, ED, OV, DF, DR, OE, LD and FD are calculated for similarity comparison. Assume that using AT1 in the AAP we get: DD = 10%, ED = 25%, OV = 10%, OE

= 5%, LD = 15%, FD = 15%, DF = 5%, DR = 50%. The degree of similarity is derived as:
s(x) = (4 + 3 + 4 + 5 + 3 + 3 + 5 + 0) / (8 x 5) = 27/40 = 67.5%.

Assume using AT2 in the AAP we get: DD = 10%, ED = 10%, OV = 5%, OE = 5%, LD = 15%, FD = 15%, DF = 5%, DR = 5%. The degree of similarity is derived as:
s(x) = (4 + 4 + 5 + 5 + 3 + 3 + 5 + 5) / (8 x 5) = 34/40 = 95%.

In such a situation we will choose AT2 as the image for the application and thus the recorded algorithm for AT2 (ALT[AT2]) will be chosen for the application. There are two possibilities. One is that the algorithm is the same as the initial one selected for this application so that there is no improvement to be made by the tuning and the results are displayed at the levels of details requested; the other is that the algorithm is changed to a new one. The data is reorganised according to the new algorithm and the degree of similarity, algorithm and the data set is recorded by the system in SRi rules. If there is no ATi in the AAP that matches the application then heuristic rules will be applied to choose an algorithm. To use these rules the properties of d(even), Rs, Dyn, |Ds| and Dxy are calculated. An example has been given in 4.4. A logical picture for the example is illustrated in 4.36.

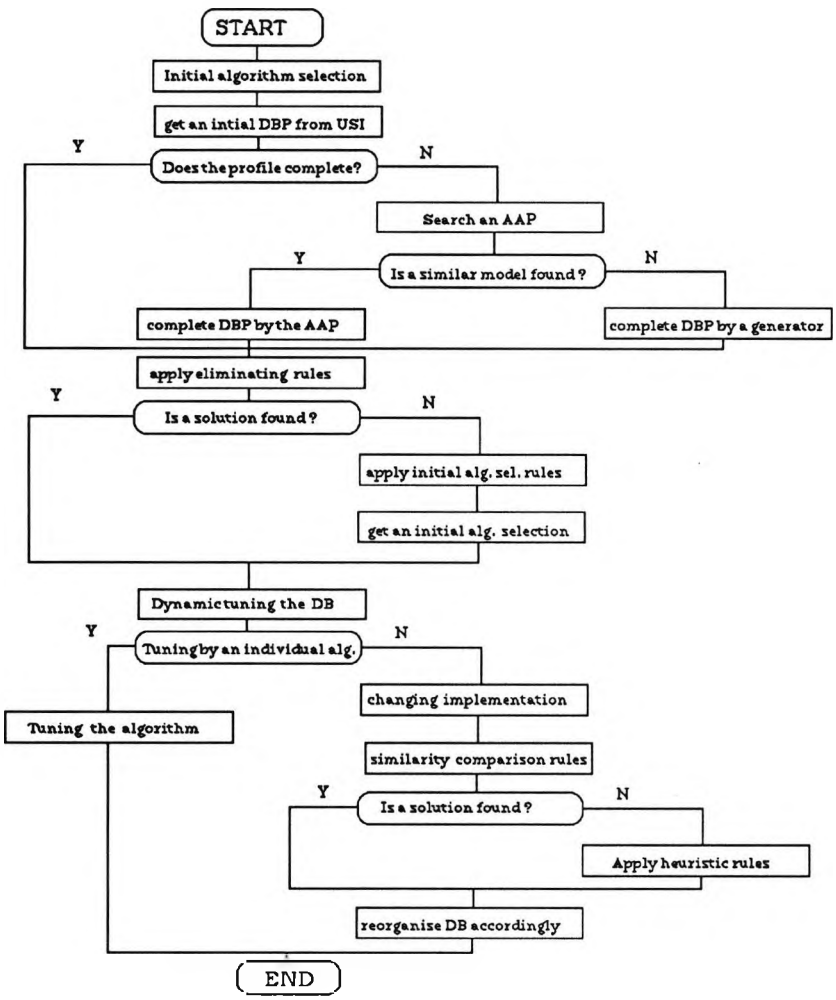


Figure 4.36. An illustration for an example.

From this example we realise that:
" The choice of a representation is not clear cut. It depends on the problem domain and the task to be performed. "

4.9 Expert system tuning and validation.

A flexible deductive inference is desirable in all knowledge base systems. The reason for this is the nature of knowledge is evolutionary and no perfect solution exists for problems that involve decision making. Particularly in dynamic situations, human knowledge is under development throughout problem- solving process. However, a knowledge system implies a strategy to improve the ability to solve problems and to derive a solution as valid as possible and at the same time, to adapt to a varying environment. For instance, in a fast-moving, competitive environment, a high accuracy may have to be traded for speed; and high performance may be traded for simple implementation for a short-lived data set. This is performed by application of knowledge to a problem to be tackled and continuous adjustment according to specific circumstances. Knowledge can thus be refined during system evolution for more accurate inference to problem-solving. This is done by assessing the deviation between the results achieved by system inference and those expected. In this section, a brief literature survey is presented. The main focus is on tuning and validating the expert system for m-d physical database design. However, tuning and verification of an expert system is a project in its own right. As a result, we only discuss specific aspects of tuning and verifying our expert system.

4.9.1 Introduction

Knowledge base systems are attracting more and more attention from the I.T. industry in recent years. The viability, however, of an expert system depends very much on its cost against the benefit, user-system interface and validity. Especially for expert systems, the problem-solving process applies heuristics which feature inexact inference, educated guesses, experience, instinct of an expert, and approximate evaluations. This feature of expert systems demands a different verification approach from that of non-expert systems. Very often, statistical information is collected for evaluating and verifying an expert system. In this section, we mainly examine the validity of an expert system. A lot of research work has been done in the field of validation of non-expert systems [BE89a] [BE89b]. However, not much has addressed for the validation of expert systems [WE83]. The salient difference between a non-expert system and an expert system is

determined by the system nature. Generally speaking, the former has static and definite features in respect to getting results - the knowledge to solve a problem is embedded with procedures; whereas the latter possesses dynamic and uncertain characteristics in terms of obtaining solutions - the knowledge for problem-solving is treated as data which can be changed during processing and that can influence the expected results (eg. the alteration of a quantity can lead to changes of problem nature, alteration in environment and goals can affect the inference paths). In the author's opinion, to validate a non-expert system the emphasis is based on an application of all possible DD-paths (Decision-to-Decision paths) which are likely to occur in various runs. In addition to specific application knowledge the inference approach is a kernel element in validating an expert system, i.e. how a solution is derived has to be examined. The inference approach needs to apply the new knowledge obtained from a dynamic environment. Furthermore, an element which resolves the inaccuracy of reasoning, i.e. the system tuning, has to be introduced in an expert system to improve the accuracy during the live system cycle. This tuning is based on known and machine-learned knowledge.

4.9.2 Literature survey

Validating and tuning/refining expert systems has been examined by several authors. However [WE83], the two approaches are referred to as : (1) the anecdotal approach and (2) the empirical approach. The former evaluates a program by its cases. If a specific program performs poorly, attempts are made to correct the program which can cause problems, since unused cases can be affected by this change. The latter places emphasis on the empirical evaluation over many problem cases. Using this approach, many cases have to be generated to test the system, i.e.this provides enough representative coverage for validation. Considering reliability of expert systems, the author [HO89b] suggested various criteria in an expert system evaluation. These criteria include: (1) correctness of final decision, (2) accuracy of final decision, (3) correctness of reasoning techniques, (4) sensitivity analysis, (5) robustness, (6) quality of USI and (7) cost effectiveness.

4.9.3 Expert system tuning

This section focuses on tuning and verification of m-d physical DB design expert systems. Tuning in m-d physical DB design is divided into two levels:

(1) modifying existing knowledge;

(2) adapting inference rules to modified knowledge and adding constraints to

refine rules.

Tuning performance for a specific application is carried out based on specific knowledge, which is collected dynamically. It concerns response time (speed) and storage utilisation features for performance. To apply heuristics and reduce system overhead, the feature values at the performance tuning points, i.e. dramatic changes in performance, are stored.

System tuning considers the performance of the expert system. The major factors considered here are reasoning validity and correctness of a final decision. The former improves inferences behind various rules; the latter enhances the success rate of the whole system.

4.9.3.1 General considerations

There are two major aspects in which an expert system can be validated and verified: (1) behaviour of an expert system, (2) ontology of an expert system. Behaviour covers granularity, capability, correctness, optimality and USI issues; while ontology includes structure suitability, consistency, validity, completeness and accuracy issues. As an experimental expert system, the focus is on ontological issues.

The experimental KBS is constructed based on heuristics. Consequently, accurate terms for correctness of the system cannot be defined, i.e. the tuning and validating process itself has to apply heuristics. In this system, a heuristic approach is based on historical data and the system evolution process. The data records summary information about individual application cases and stores this information in the knowledge base for rule performance analysis. The outcome of this analysis enables the validation and verification of the system in order to achieve better results. The system evolution process analyses various factors which may influence decisions, and with additional historical data, the performance of the reasoning process can be further improved. Similar patterns are extracted and cluster analysis is applied to perform the task. In this section we limit ourselves to specific aspects of the tuning and validation process, as this topic alone could be a project in itself.

4.9.3.2 Information about rules in general

To verify an inference process, the summary information about application cases in which rules are applied successfully, is recorded. This information is kept by relating the inference process to performance, recording the rates on the number of successful instantiation and the total number of all instantiation. In addition, the inference structures of failure cases are also stored. This information is employed to analyse the validity of the rule base. The logical structure of tuning and verification is illustrated in Figure 4.37.

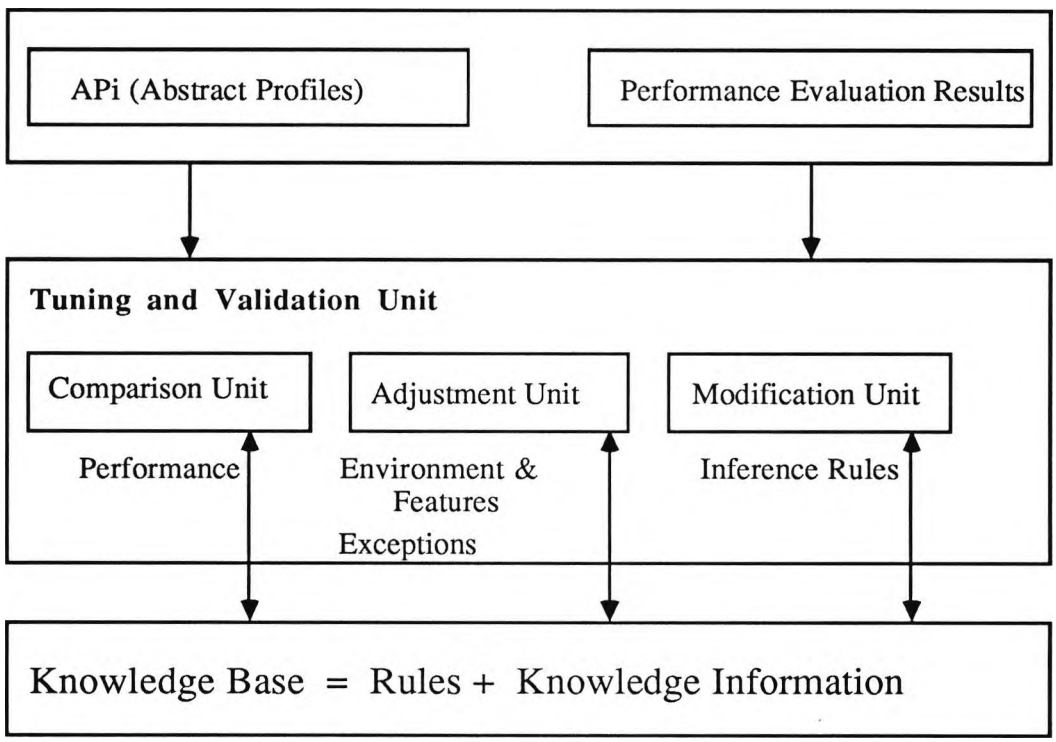


Figure 4.37 Logical structure of tuning and validation.

The diagram illustrates that tuning and validation mainly takes inputs from abstract

profiles (data set profiles) and performance results from each individual profile. Logically all these abstract profiles that lead to the same solution (or falling in the same performance spectrum) are linked (data set uses the same access algorithms) together, providing a basis for the comparison unit. The comparison unit is applied for tuning and validation. Information kept for profiles is logically classified into salient features, and application environment. The result of this comparison is used to adjust heuristics and rules concerning dynamic features. The adjustment can be done either by updating boundaries of various feature spectrums which form the basis of a particular solution; or by modifying rules set to include new constraints.

Comparison unit

In our particular situation, the comparison unit takes profiles from those applications which employ the same algorithm and ranks their performance. This task is straightforward. The ranking, as a result from the comparison unit, then relates to individual profiles which the adjustment unit can pursue further. Alternatively, the comparison unit takes performance criteria and the performance of applications as inputs, analyses features of applications which show satisfied results, and passes the analysis to the adjustment unit.

Adjustment unit

The adjustment unit separates features for a group of profiles, examining the difference between them, producing exceptions and making necessary adjustments by using expert knowledge (heuristics). For instance, feature 1 and feature 2, according to the system knowledge, are two correlated features with different weights. Analysing their values in these k profiles can produce suggested modifications on weights. For the profiles considered, we have:

	weight-1	feature-1	weight-2	feature-2	
profile-1	w11	f11	w12	f12	(best performance)
profile-2	w21	f21	w22	f22	
.					
.					
.					
profile-k	wk1	fk1	wk2	fk2	(worst performance)

As the system knows the meaning of these ranks (weights) among these profiles, it heuristically adjust the value of w21 to wk1, w22 to wk2 to that of w11, w12 for f11, f12. The system adjusts according to the following heuristics, in which the w11 and f11 is used as criteria for other profiles:
(1) from w11 / w12 = wz1 / wz2 where z = 2, 3, ..., k, we get adjusted values:

$$wz1^{(a)} = \frac{w11 \times wz2}{w12}$$

$$wz2^{(a)} = \frac{wz1 \times w12}{w11}$$

where $wz1^{(a)}$ or $fz1^{(a)}$ is the result from adjusting the weight or boundary.

Consequently, if in due course, a better performance is obtained from another group of profiles, this adjustment process can be repeated. The same principle applies to the results gained from analysis of those satisfied applications.

During the above described process, if however, a contradictory situation occurs, exception rules are to be added to cope with this situation. Adding exceptions to the rule base involves human intervention and analysis.

Modification unit
The modification unit performs an update operation over the knowledge and rules based on the results from the adjustment unit. Weights and boundaries may change and exception rules can be added. Modification and adjustment can also be performed by tuning individual features (e.g. expanding boundary set, and applying high resolution) when required.

4.9.3.3 Feature information

Based on stored historical data, modification can be done for a specific feature over its spectrum. For example, a feature spectrum of a profile is initially set to be F = {f1, f2, ..., fm}, a solution set for a specific problem domain is S = {s1, s2, ..., sn}. Here fi, where i ∈ 1, 2, ..., m, is a feature spectrum, representing a range from, say, a to b.

Each i feature fi ∈ F is mapped to the solution set, say, fi -> sj. Since F is set by applying heuristics, the mapping fi -> sj can have f'i -> sj as an improved version to the initial setting, i.e. tuning is aimed at achieving better solutions. The tuning process can be described as follows:

the goal - modify fi -> sj to be f'i -> sj;

where the latter has better performance;

method - obtain historical data within spectrum fi

fi = {fi1, fi2, ..., fix};

fi is obtained by ordering values within the range from a to b, i.e. extract fi1, fi2, ..., fix from APs in the knowledge base, calculate performance for each value of fi:

pi = {pi1, pi2, ..., pix};

or alternatively, get performance from the knowledge base if they are already recorded and work out the optimal one among pi optimal(pi) = pij, j ∈ 1, 2, ..., x replace fi with fij, i.e. set f'i = fij.

Modification can also be done on a set of features. This is done by grouping high correlated features together to work out the effects. The tuning process is as follows:

the goal - replace fi -> sj with f(I, J) x fi -> sj

I and J are other features used to present the profile respectively.

where the latter achieves better results

method - obtain historical data for I, J to get the spectrum value:

I = {I1, I2, ..., Ix}

J = {J1, J2, ..., Jy}

evaluate influence of I, J

(1) p(I, fi) = {pi1, pi2, ..., pix}

(2) p(J, fi) = {pi1, pi2, ..., piy}

work out the effect of I and J;

replace fi with f(I, J) x fi.

where f(I, J) is a correlation coefficient. It is calculated by referring to performance, i.e. when a summation is used as the heuristic to score a preference, a lower score will be given to I if J happens to have a tendency to deteriorate the performance and thus the heuristic is used to reduce the total scores (f(I, J) = 0.8 -> f(I, J) = 0.5, for instance). The negative accumulation effect is, therefore, considered.

4.9.3.4 Information about applications

As mentioned above, feature values at performance turning points are kept for an application. Initial feature information is compared and analysed in order to work out what are the influential factors for performance change. This analysis is then stored as knowledge for later use. If incomplete information is supplied, instead of ceasing the system function, the expert system guesses the missing bit by referring to the knowledge stored in the system. Alternatively, it generates information for the missing part using its best knowledge. This is done by accessing a similar application stored in the knowledge base to provide a good guess.

4.9.3.5 Extract information for similar applications

Extracting information for similar applications is based on the solution, i.e. the access algorithm employed. If applications use the same algorithm then they will be categorised as the similar applications. We can show this in Figure 4.38. This information will later become part of the knowledge base, which derives an abstract application profile.

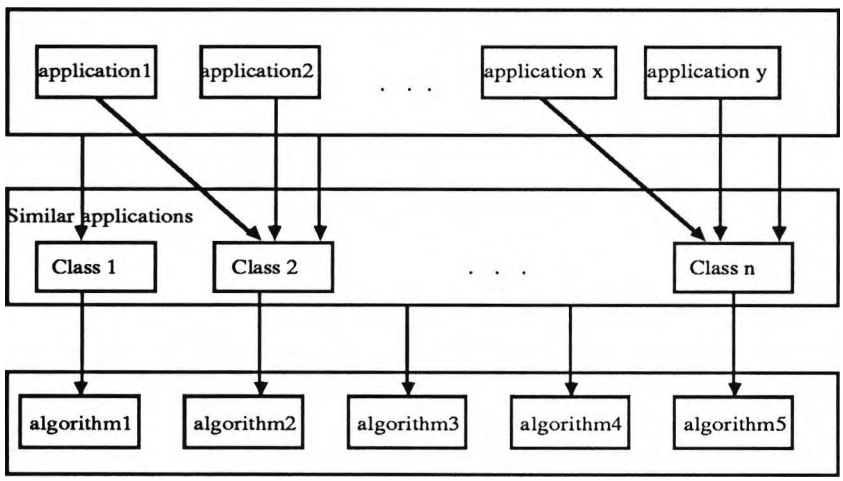


Figure 4.38 Similar application.

This diagram indicates that a bitmap can be effectively utilised to extract information from similar applications. This data structure is illustrated in Figure 4.39.

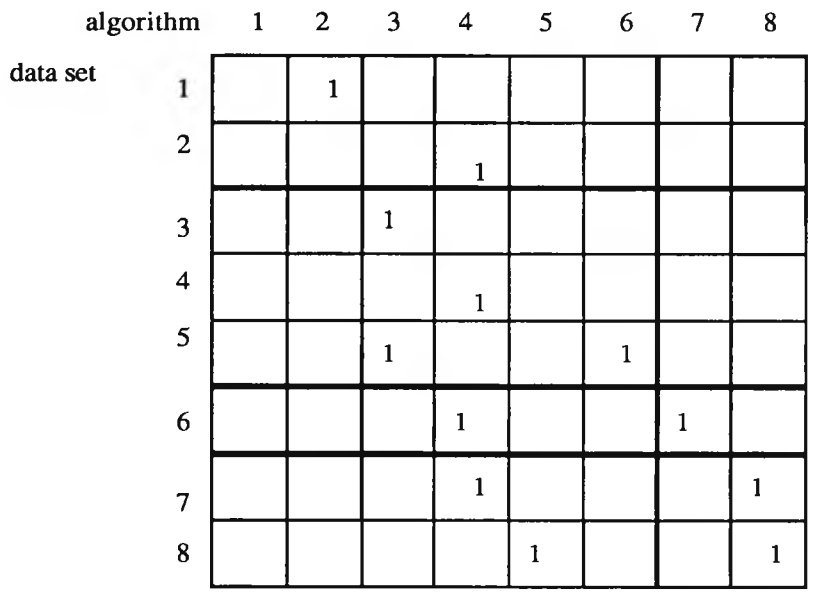


Figure 4.39 Bitmap for similar applications.

In the bitmap, each bit of the row indicates an application processed by the system. Each bit in the column corresponds to a selected access algorithm. The position of a bit tells which algorithm is employed for a specific data set.

4.9.3.6 The impacts over change of search space size

Partitioning strategies are influenced by dimensionality. As mentioned before, partitioning mainly affects storage utilisation. When dimension increases, the storage utilisation can be decreased under a m-d partition. As a result, it is desirable to control the increase of search space dimensionality. This can be handled by distinguishing the granularity of various features. The granularity here means the size of the domain for

the feature concerned. The dimension with a small sized domain can be dropped from the search space if dimensionality increases unexpectedly. A typical relational database can have a large number of relations of 1-d and 2-d search space; medium number of relations of 3-d and 4-d search space; and a small number of relations with higher m-d ($m > 4$) search space.

4.9.3.7 The impact over the domain size

As mentioned above the size of domain can play a significant role in tuning the physical database design. The size of domain of a particular feature can also affect access paths, which can be formed dynamically. From experience, when a retrieval is conducted on several attributes (that is what the m-d search space is designed for) with different size of domains, it is obvious that tuning effort should be given to a large domain. The size of a domain is the number of different values in it, i.e. if domain Di = {Di1, Di2, ..., Dix} then the size of the domain is |Di| = x. This feature depends on the individual data set. The reason is that the large domain is likely to have more splits than that of the smaller ones. Therefore, it tends to become a dominant dimension. An illustration is given in Figure 4.40.

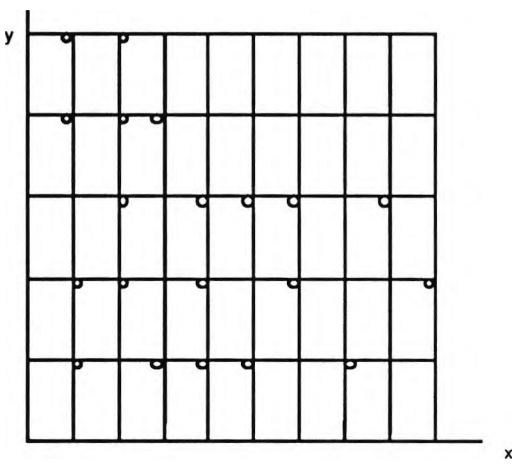


Figure 4.40 A 2-d search space.

In this diagram, dimension x has a larger domain than that of dimension y. The x dimension is the dominant feature in this 2-d search space.

4.9.4 The results of tuning analysis

The system tuning concerns mainly two aspects: (1) knowledge information; (2) inference process. The first aspect determines the relevant category of profiles; the second one constructs the reasoning process based on the knowledge information.

In the consideration of the physical database design, knowledge information is allowed to be modified by a refining process. This refining process gathers feature information from various applications and evaluates the performance of the current knowledge to produce modifications to improve the success rate of the system. In addition, the reasoning process is adaptable to the modified knowledge information.

Example

Modifying knowledge information.

Originally, a data distribution can be divided and measured by the following heuristics:

Measurement - D(even)

Ddis = { d1, d2, d3, d4, d5 }

= { even, fairly even, relatively even, not even, uneven }

= { < 15%, < 25%, < 40%, < 50%, > 75% }

When selecting an algorithm, considering storage utilisation, we should consider the size of the data set because the size is correlated with the data distribution. As a result, a large sized data set may trigger a modification over the knowledge information. This modification adds a multiplicity factor to the Ddis measurement and refines the meaning of varied categories of data distribution. For this particular circumstance, we first assign the factor as 0.8. And therefore, we have:

Ddis = { < 12%, < 20%, < 32%, < 40%, > 60% }.

Similarly, if a smaller sized data set is the case, the factor can be assigned as 1.2, which results:

Ddis = { < 18%, < 30%, < 48%, < 60%, > 90% }.

The reasoning process is adapted to suit this change by adding a decision rule, which judges the size of a data set concerned, to refine data distribution measurement. This is done by considering correlated factors of data distribution, i.e. data set size.

This example is over-simplified for illustration. To refine knowledge information, decisions need to be made on historical information stored for various applications, especially those with unsatisfactory results.

4.9.5 Examples

Information about performance (access speed and storage utilisation) and the reasoning process require to be stored for tuning the system heuristics. The result of this tuning is employed to adjust rules and knowledge information. The framework is shown in Figure 4.41, which is based on Figure 4.37. Several examples are also presented.

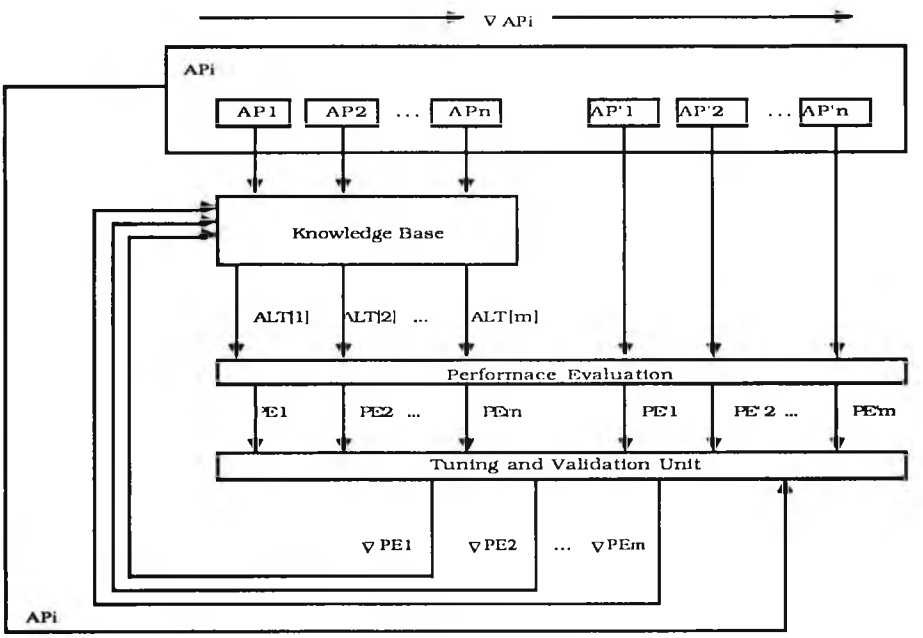


Figure 4.41 The Framework for System Tuning.

Here

APi - initial application profile for application i, for i = 1, 2, ..., n.

AP'i - application profile after changes made to the original ones.

∇Api - differences between APi and AP'i.

ALT[i] - algorithm i, for i = 1, 2,..., m.

PEi - performance evaluated for application profile APi, for i = 1, 2, ..., m.

PE'i - performance evaluated for application profile AP'i.

∇PEi - differences between PE'i and PEi (i.e. PEi - PE'i)

In this framework the profile about those applications which apply the same access algorithm is recorded. The profile is then used to evaluate the performance. The heuristic rule is tuned based on the differences in the performance.

Example 1 - tuning by heuristics gained from different types of applications:

Suppose application "A" has the following features:

Deven $\leq$ 20%

Rs $\leq$ 30%

Dyn $\leq$ 20%

Ps $\leq$ 30%

|Ds| = 3 (data set size rating)

and application "B" has the following features:

Deven $\leq$ 30%

Rs $\leq$ 30%

Dyn $\leq$ 20%

Ps $\leq$ 40%

|Ds| = 5 (data set size rating)

According to the heuristic function in section 4.4.3 the result calculated for application "A" is:

$h_{EXCELL} = 4 + 5 + 3 + 3 + 3 = 18$

$h_{z\text{-hashing}} = 5 + 3 + 4 + 3 + 4 = 19$

$h_{quantile\text{-hashing}} = 3 + 3 + 4 + 3 + 3 = 16$

$h_{PLOP\text{-hashing}} = 3 + 4 + 2 + 5 + 3 = 17$

$h_{BANG\text{-}file} = 1 + 3 + 2 + 3 + 3 = 12$

For application "B" the calculated results are:

$h_{EXCELL} = 5 + 5 + 3 + 2 + 1 = 16$
$h_{z\text{-}hashing} = 4 + 3 + 4 + 4 + 3 = 18$
$h_{quantile\text{-}hashing} = 4 + 3 + 4 + 5 + 4 = 20$
$h_{PLOP\text{-}hashing} = 4 + 4 + 2 + 3 + 4 = 17$
$h_{BANG\text{-}file} = 2 + 3 + 2 + 1 + 5 = 13$

As a result, access algorithm "z-hashing" is selected for application "A" and "quantile-hashing" is chosen for application "B". However, as data are added to, or deleted from, application "B" the feature Deven is changed to be within the spectrum of $\leq 20\%$, thereby approaching feature of application "A". At this point, if the performance of application "A" is worse than that of application "B", then an adjustment can be introduced.

Assume after dynamic changes to application "B" the features are near to those of "A", and become:

Deven $\leq 20\%$
Rs $\leq 30\%$
Dyn $\leq 20\%$
|Ds| $= 4$ (data set size rating).

For those features the performance measured for application "B" is better than application "A".

Say, for those two applications we have the following results:

A: point search	:	Tp	= 2 x Tsec
storage utilisation	:	Su	= 65%
range search accuracy	:	a	= 65%
B: point search	:	Tp	= 1.5 x Tsec
storage utilisation	:	Su	= 70%
range search accuracy	:	a	= 50%

This indicates that quantile-hashing is a better choice under the initial condition of application "A" than that of z-hashing. As a result, we can change the heuristic rule

by increasing its weights to Deven for the quantile-hashing algorithm, i.e.

Deven = {≤ 20%, ≤ 25%, ≤ 40%, ≤50%, ≥ 50% }

$h_{d(even)}$(quantile-hashing) = (3, 4, 5, 2,1) can be refined to be:

$h'_{d(even)}$(quantile-hashing) = (6, 8, 10, 4, 2)

Here

$h'_{d(even)}$(quantile-hashing) is derived by the following calculation:

$h'_{d(even)}$(quantile-hashing) = $\lceil p/q \rceil$ x $h_{d(even)}$(quantile-hashing)

where p = 5 is the weight assigned to z-hashing for Deven ≤ 20%,

q = 3 is the weight given to quantile-hashing before changing for Deven ≤ 20%.

After refining the rule, the initial selection for application "A" becomes:

h_{EXCELL}	= 4 + 5 + 3 + 3	= 15
$h_{z\text{-hashing}}$	= 5 + 3 + 4 + 4	= 16
$h_{quantile\text{-hashing}}$	= (6) + 3 + 4 + 3	= 16
	where (6) is value given after refinement	
$h_{PLOP\text{-hashing}}$	= 3 + 4 + 2+ 3	= 12
$h_{BANG\text{-file}}$	= 1+ 3 + 2 + 3	= 9

It means that quantile-hashing has become one of the candidates to be chosen. This is an over-simplified example. Here the refinement is based on heuristic learning, i.e. the system learns from the performance evaluation and updates the initial rule for a better initial selection. In real situations, however, the system tuning needs human intervention, especially at the beginning. The system gives comprehensive data to users, to assist them in analysing the interrelation between various factors which influence the decision. As a result, this relationship can be built into the system learning mechanism so that less human intervention is required later on.

Example 2 - tuning by evaluating performance for an application

As shown in Figure 4.42 we assume the following features.

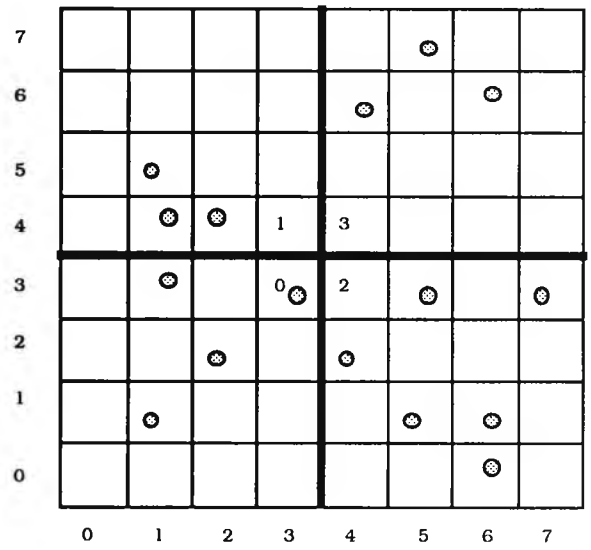


Figure 4.42 2-d search space for a sample data set (original).

Data distribution

b = 4

Ds = { (1,1) (1,3), (1,4), (1,5), (2,2), (2,4), (3,3), (4,2),
(4,6), (5,1), (5,3), (5,7), (6,0), (6,1), (6,6), (7,3) }

the equivalent z-code for the data set is:

Ds(z-code) = {3, 7, 18, 19, 12, 24, 15, 36, 52, 35, 39, 55, 40, 41, 60, 47}

and z-order for the data set is:

Ds(z-order) = { 3, 7, 12, 15, 18, 19, 24, 35, 36, 39, 40, 41, 47, 52, 55, 60 }

The data distribution can be derived from the following steps:

(1) derive the number of data points in each grid cell

From z-code the equal grid cell partition generates a boundary set d, d = { 15, 31, 47, 63}. Here d divides the search space into four partitions and each

partition has its own data items:

G[0] = {3, 7, 12, 15}

G[1] = {18, 19, 24}

G[2] = {35, 36, 39, 41, 47}

G[3] = {52, 55, 60}

Hence the size of each grid is derived as:

| G[0] | = 4 | G[1] | = 3

| G[2] | = 6 | G[3] | = 3

(2) calculate resolution

r = |Ds| / b = 16 / 4 = 4

(3) calculate Deven

$$Deven = \frac{\sum_{i=1}^{i=r} |C[i] - b|}{b \times r} = \frac{4}{4 \times 4} = 25\%$$

Query

We use x and y to represent two different key attributes since the dimensionality of this example is two (m = 2). The query frequency is then assumed to be:

query on the dimension x

Qx ≤ 40%

query on the dimension y

Qy ≤ 60%

point search by involving both dimensions

Ps ≤ 30%

range search on the dimension x by changing y (fixed x value)

Rs(x) ≤ 20%

range search on the dimension y by changing x (fixed y value)

Rs(y) ≤ 20%

range search involving both dimensions:

Rs(x,y) ≤ 30%

Dynamic features

Let us assume that dynamic features follow a time sequence, i.e. they change during the life of a data set.

Case	Deletion rate	Insertion rate
(1)	18.75%	12.5 %
(2)	18.75%	18.75%
(3)	25 %	50 %
(4)	25 %	62.5 %

First stage (initial features)

According to the given features, the following factors are considered:

Deven	= 25%	(measured under the condition r = 4)
Dyn	≤ 35%	(estimated at initial stage)
Rs	≤ 70%	(estimated at initial stage)
Ps	≤ 30%	(estimated at initial stage)
|Ds|	= 3	(estimated at initial stage)

Employing heuristic function described in section 4.4.3 we have:

h_{EXCELL} = 5 + 2 + 2 + 1 + 3 = 13
$h_{z\text{-hashing}}$ = 4 + 4 + 2 + 5 + 4 = 19
$h_{quantile\text{-}hashing}$ = 4 + 4 + 2 + 4 + 3 = 17
$h_{PLOP\text{-}hashing}$ = 4 + 2 + 4 + 4 + 3 = 17
$h_{BANG\text{-}file}$ = 1 + 4 + 5 + 1 + 3 = 14

Hence z-hashing algorithm is initially selected.

Performance evaluation

Point search average speed Tp

(1) With overflow handling (assuming only when n > 1.5 x r x b next split will be triggered and the resolution r = 4)

$$Tp = \frac{\sum_{i=0}^{i=r-1} T(G[i])}{r} = \frac{4 + 3 + 8 + 3}{16} = 1.125 \times Tsec$$

Here T(G[i]) is the total number of secondary storage accesses required by

individually accessing each data item (a point) in grid cell [i]. For example, there are six points in the grid cell 2. Assuming four points are stored in home bucket and two points are stored in the overflow bucket. The total number of disk accesses is calculated as 4 + 2 x 2 = 8, where the first item 4 is the number of accesses for home points and the second item 2 x 2 = 4 is the number of accesses for the overflow points.

(2) Without overflow handling the hashing algorithm always needs 1 access to locate the required data at the cost of storage space. As a result the calculation is insignificant.

Range search accuracy a

(a) select (x,y) where $x \leq 5$ and $y \geq 5$

$$a = \frac{\text{number of points satisfy search condition}}{\text{total number of buckets searched x b}} = \frac{2}{8} = 25\%$$

(b) select (x,y) where $2 \leq x \leq 6$ and $y \leq 4$

$$a = \frac{\text{number of points satisfy search condition}}{\text{total number of buckets searched x b}} = \frac{6}{16} = 37.5\%$$

(c) average of case (a) and (b)

$a = (2/4 + 6/16)/2 \approx 31.25\%$

Note: in a real situation a great number of cases will be utilised to calculate the average range search accuracy.

Storage Utilisation

(1) With overflow handling (r = 4)

minimal requirement $Nmin = \lceil n/b \rceil = 16 / 4 = 4$ blocks

actual requirement $Nmax = \sum_{i=0}^{i=r-1} \lceil G[i]/b \rceil = 5$ blocks

$Su = Nmin/Nmax = 4/5 = 80\%$

(2) Without overflow handling (r =8)

minimal requirement $= \lceil n/b \rceil = 16 / 4 = 4$ blocks

actual requirement $= N - Nempty = 8 - 0 = 8$ blocks

here N is the number of grid cells required for z-hashing without overflow handling.

Su = 4/8 = 50%

Second stage (introduce changes)

Case (1)

From the original data set delete (4, 6), (6, 6), (7, 3) and inset (2, 7), (3, 5) we have the following diagram:

Case (1)

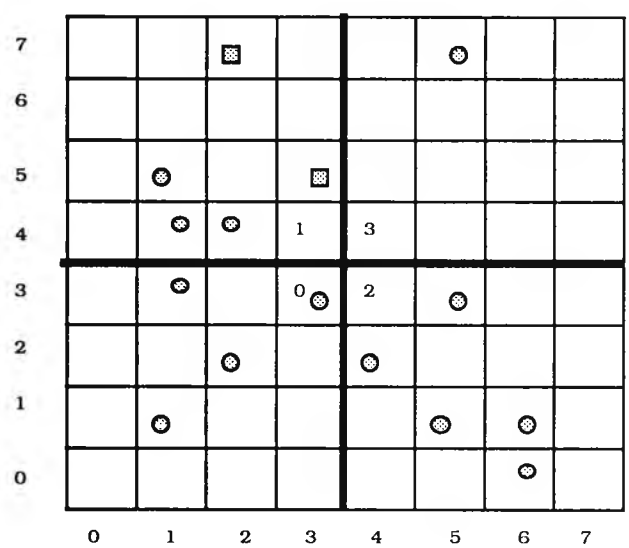


Figure 4.43 The 2-d search space after 18.75% deletions and 12.5% insertions (the squares are inserted data points and the circles are original data points).

Using the same method as in the initial case for *Case (1)* the following Deven and performance information are derived:
Deven ≈ 33.3%

Performance evaluation

Point search average speed Tp

(1) With overflow handling (assuming only when n > 1.5 x r x b next split will be triggered and the resolution r = 4)
Tp ≈ 1.13 x Tsec.

Range search accuracy a

(a) Select (x,y) where x ≤ 5 and y ≥ 5
a = 25%

(b) Select (x,y) where 2 ≤ x ≤ 6 and y ≤ 4
a = 37.5 %

(c) Average of case (a) and (b)
a = 31.25 %

Storage Utilisation Su

(1) With overflow handling (r = 4)
Su ≈ 66.7 %

(2) Without overflow handling (r =16)
Su = 4/8 = 50%

Case (2)

From the original data set delete (2, 2), (3,3), (4,6) and inset (3, 7) (5, 6), (7, 1) we have the following diagram:

Case (2)

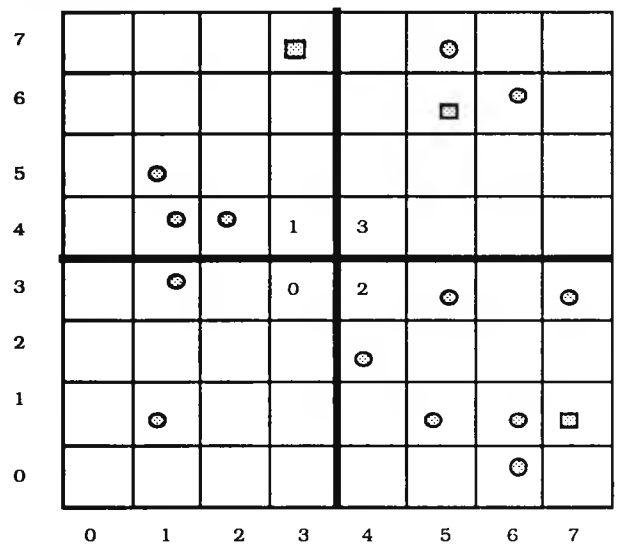


Figure 4.44 The 2-d search space after 18.75% deletions and 18.75% insertions .

After the change Deven becomes: $Deven = \frac{\sum_{i=0}^{i=r-1} |C[i] - b|}{r \times b} = \frac{3 + 2 + 1}{4 \times 4} = 37.5\%$

Performance evaluation

Point search average speed Tp

(1) With overflow handling (assuming only when n > 1.5 x r x b next split will be triggered and the resolution r = 4).

$$Tp = \frac{\sum_{i=0}^{i=r-1} T(G[i])}{r} = \frac{4 + 2 + 3 + 10}{16} = 1.1875 \times Tsec$$

Range search accuracy a

(a) Select (x,y) where $x \leq 5$ and $y \geq 5$

a = 37.5%

(b) Select (x,y) where $2 \leq x \leq 6$ and $y \leq 4$

a = 31.25 %

(c) Average of case (a) and (b)

$a \approx 34.3$ %

Storage Utilisation Su

(1) With overflow handling (r = 4)

Su = 4 / 5 = 80%

(2) Without overflow handling (r =8)

Su = 4/8 = 50%

Case (3)

From the original data set delete (4, 6), (5, 1), (6, 6), (7, 3) and insert (0, 7), (2, 5), (3, 1), (5, 2), (5, 5), (6, 3), (6, 5), (7, 2). See Fig 4.45 for *Case (3).*

Case (3)

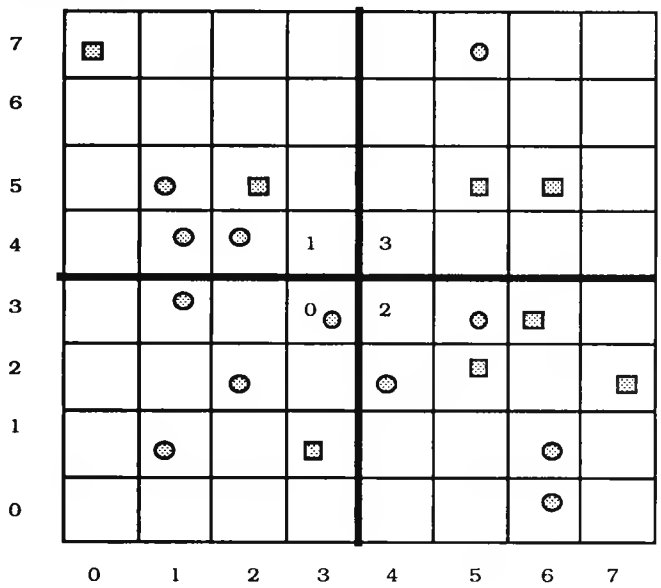


Figure 4.45 The 2-d search space after 25% deletions and 50% insertions.

Deven = 37.5%

Performance evaluation

Point search average speed Tp

(1) with overflow handling (assuming only when n > 1.5 x r x b next split will be triggered and and the resolution r = 4)

Tp = 1.25 x Tsec

Range search accuracy a

(a) select (x,y) where $x \leq 5$ and $y \geq 5$

a ≈ 16.7 %

(b) Select (x,y) where $2 \leq x \leq 6$ and $y \leq 4$
a = 40%

(c) Average of case (a) and (b)
a ≈ 28.3 %

Storage Utilisation Su

(1) With overflow handling (r = 4)
Su ≈ 71 %

(2) Without overflow handling (r =16)
Su ≈ 25 %

Case (4)
From the original data set delete (1, 1), (5, 1), (5, 3), (5, 7) and insert (0, 4), (0, 5), (0, 6), (2, 6), (4, 7), (6, 2), (6, 3), (6, 7), (7, 2), (7, 5). See Fig 4.46 for *Case (4)*.

Case (4)

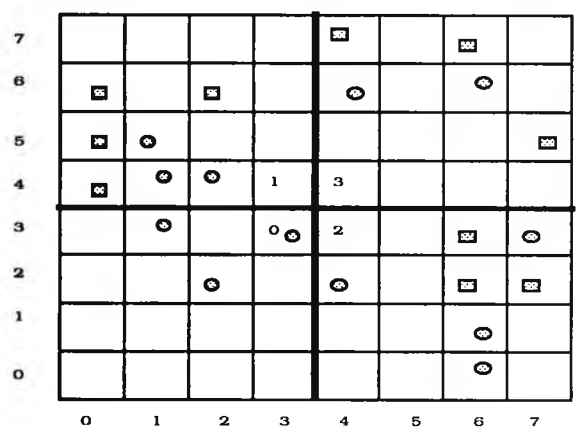


Figure 4.46 The 2-d search space after 25% deletions and 62.5% insertions.

Deven = 50%

Performance evaluation

Point search average speed Tp

(1) with overflow handling (assuming only when n > 1.5 x r x b next split will be triggered and and the resolution r = 4)

Tp ≈ 1.32 x Tsec

Range search accuracy a

(a) Select (x,y) where $x \leq 5$ and $y \geq 5$

a ≈ 33 %

(b) Select (x,y) where $2 \leq x \leq 6$ and $y \leq 4$

a = 25 %

(c) Average of case (a) and (b)

a ≈ 29.1 %

Storage Utilisation Su

(1) With overflow handling (r = 4)

Su ≈ 86 %

(2) Without overflow handling (r =16)

Su ≈ 37.5 %

Analysis of performance results

Case	Deven	Tp	average accuracy	Su(O)	Su(W)	Dyn
(0)	25 %	1.125	31.25%	80 %	50 %	
(1)	33.3%	1.13	31.25%	66.7%	50 %	21.25 %
(2)	37.5%	1.1875	34.3 %	80 %	50 %	37.5 %
(3)	37.5%	1.25	28.3 %	71.3%	25 %	75 %
(4)	50 %	1.32	29.1 %	86 %	37.5%	87.5 %

Here Su(O) is the storage utilisation with overflow handling; and Su(W) is storage utilisation without overflow handling.

(1) Feature measurement refinement

The performance shows that the initial feature measurement for Dyn is not accurate. A better measurement for dynamic situation Dyn should be Dyn = |Ir - Dr|. Therefore the rule for calculating Dyn is modified from Dyn = Ir + Dr to Dyn = |Ir - Dr|. The results are changed to be:

Case	Deven	Tp	average accuracy	Su(O)	Su(W)	**Dyn (after change1)**
(0)	25 %	1.125	31.25 %	80 %	50 %	
(1)	33.3%	1.13	31.25 %	66.7%	50 %	**6.25 %**
(2)	37.5%	1.1875	34.3 %	80 %	50 %	**0 %**
(3)	37.5%	1.25	28.3 %	71.3%	25 %	**25 %**
(4)	50 %	1.32	29.1 %	86 %	37.5 %	**37.5 %**

As Dyn relates to Deven, it is found that the Dyn value gained from each individual grid cells is more valid than if gained from the entire search space. That is, Dyn can be further refined as:

$$Dyn = \sum_{i=0}^{i=r-1} |Ir[i] - Dr[i]| / (r \times b)$$, where Ir[i] and Dr[i] are insert rate and delete rate for grid cell i. The results after this refinement for case (4) are:

Case	Deven	Tp	average accuracy	Su(O)	Su(W)	**Dyn (after change2)**
(0)	25 %	1.125	31.25 %	80 %	50 %	
(1)	33.3%	1.13	31.25 %	66.7%	50 %	**31.25 %**
(2)	37.5%	1.1875	34.3 %	80 %	50 %	**25 %**
(3)	37.5%	1.25	28.3 %	71.3%	25 %	**25 %**
(4)	50 %	1.32	29.1 %	86 %	37.5 %	**50 %**

The formula

$$Dyn = \sum_{i=0}^{i=r-1} |Ir[i] - Dr[i]| / (r \times b)$$, where Ir[i] and Dr[i] are insert rate and delete rate will replace the previous definition of Dyn as the result of refinement.

(2) Algorithm selection analysis

The initial selection shows very good performance in point search Tp, and range search accuracy for selected searching conditions. By examine the performance for

the above cases it can be learnt that Dyn is correlated to Deven and to the average point search performance Tp. With a different Dyn value the correlation can be derived as:

Dyn	ΔDeven	Δ Tp
21.25 %	8.3 %	≈ 0.005
37.5 %	12.5 %	≈ 0.0625
75 %	12.5 %	≈ 0.125
87.5 %	20 %	≈ 0.195

From this example the results show that the initial selection is quite reasonable. However, the performance of other algorithms can be evaluated for various situations during the dynamic changes of the data set to see if z-hashing is the best one for this application.

(3) Alternative algorithm for case (4)

Features for algorithm selection

Deven	≤ 45%
Rs	≤ 70%
Dyn	≤ 50%
Ps	≤ 30%
|Ds|	= 3

Heuristic functions (refer to section 4.4.3)

h_{EXCELL}	= 2 + 2 + 3 + 3 + 3 = 12
$h_{z\text{-hashing}}$	= 2 + 4 + 2 + 3 + 4 = 15
$h_{quantile\text{-}hashing}$	= 2 + 4 + 2 + 3 + 3 = 14
$h_{PLOP\text{-}hashing}$	= 2 + 2 + 4 + 5 + 3 = 16
$h_{BANG\text{-}file}$	= 2 + 4 + 5 + 2 + 3 = 16

According to the heuristic function, either PLOP-hashing or BANG-file algorithm can be applied. Here we notice that BANG-file was given a greater weighting for the Dyn feature than that of quantile-hashing (which is the major changing factor for performance deterioration). As a result, BANG-file algorithm is chosen.

Performance evaluation for BANG-file algorithm

For *Case (4)* there are 22 points in the search space. We have assumed b = 4.

Therefore 6 partitions can be applied or 4 partitions with overflow handling. The BANG-file partition is pictured in Figure 4.47.

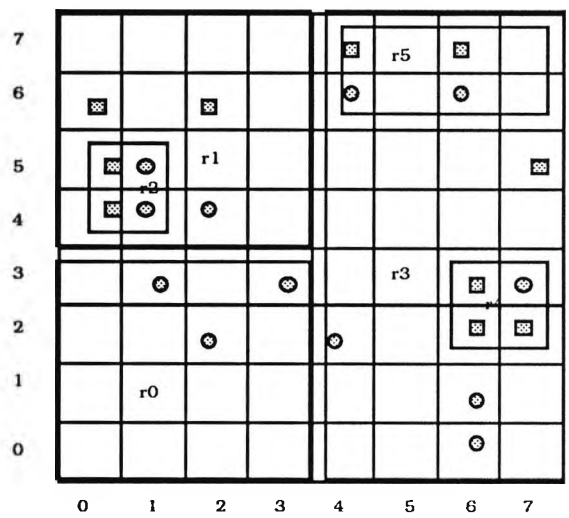


Figure 4.47 BANG-file Partition for *Case (4)*.

Region	Level	number	range (z-code equivalent)
r0	2	0	0 ~ 15
r1	2	1	16 ~ 31 - 16 ~ 19
r2	4	4	16 ~ 19
r3	2	2	32 ~ 63 - 44 ~ 47 - 52 ~ 63
r4	4	11	44 ~ 47
r5	3	7	52 ~ 63

Point search rate

Tp = 1x Tp (index in memory)

Tp = 1.5 (index on secondary storage)

Range search accuracy a

(a) Select (x,y) where $x \leq 5$ and $y \geq 5$
a $\approx$ 57% (r1 and r5 are accessed).

(b) Select (x,y) where $2 \leq x \leq 6$ and $y \leq 4$
a = 42.857 % (r0, r1, r3 and r4 are searched).

(c) Average of case (a) and (b)
a = 50%

Storage Utilisation Su

minimal requirement $= \lceil n/b \rceil = \lceil 22 / 4 \rceil = 6$ blocks

actual requirement = number of partitions + |index| = 6 + 1 = 7

Here we assume one bucket is required to store the index. In a real situation this is a much clearer case.

Su = minimal requirement / actual requirement = 6/7 = 85.8%

(4) Applying BANG-file algorithm to the original data set

Performance evaluation for the original data set applying BANG-file algorithm is shown below. The partition can be seen in Figure 4.48.

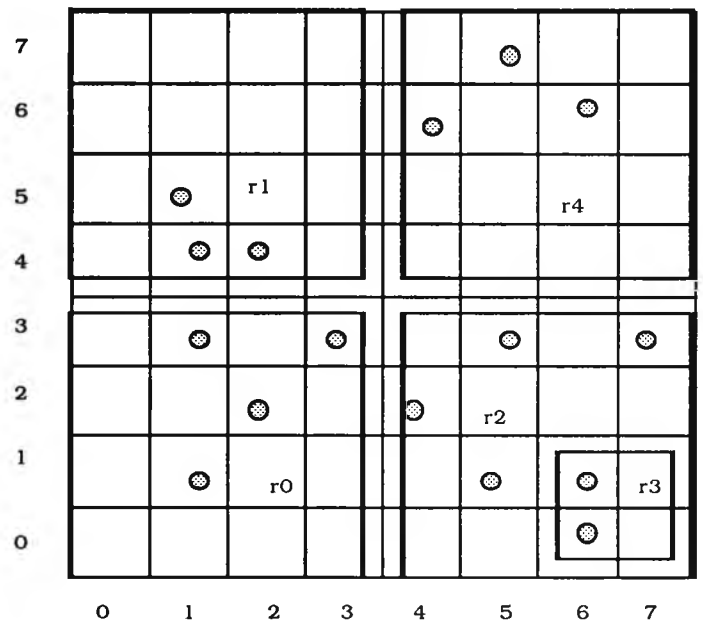


Figure 4.48 Applying BANG-file algorithm to original data set.

Point search

Tp = 1 x Tsec (index in memory)

Tp = 1.5 x Tsec (index on secondary memory)

Range search accuracy a

(a) Select (x,y) where $x \leq 5$ and $y \geq 5$
a ≈ 33% (r1 and r4 are accessed)

(b) Select (x,y) where $2 \leq x \leq 6$ and $y \leq 4$
a = 37.5 % (r0 - r5 are searched)

(c) Average of case (a) and (b)
a = 35%

Storage Utilisation Su

minimal requirement $= \lceil n/b \rceil = 16 / 4 = 4$ blocks

actual requirement = number of partitions + |index| = 5 + 1 = 6

Here we assume one bucket is required to store the index. In a real situation this is a much clearer case.

Su = minimal requirement / actual requirement = 4/6 = 66.7%

(5) comparison between initially selected algorithm and BANG-file algorithm

Algorithm	Case	Deven	Dyn	Tp	accuracy	Su
z-hashing	(0)	20%		1.125	35.0%	80.0%
BANG-file				1 or 1.5	35.0%	66.7%
z-hashing	(4)	45%	50%	1.32	30.3%	66.7%
BANG-file				1 or 1.5	50.0%	85.8%

The comparison shows that the heuristic functions are reasonably accurate. This is indicated by better performance gained for case (0) of z-hashing algorithm, the heuristically chosen algorithm; and better performance gained for case (4) of BANG-file algorithm, which is again, selected by heuristic functions based on features of case (4).

From above examples, we learnt that inaccuracy exists because the initial perception of knowledge can be inaccurate. For instance, the calculation of Dyn was based on the assumption that any changes either insertions or deletions influence the data set and the effect is measured by summation of the changes, i.e. Dyn = Ir + Dr. This measurement can be accurate when all insertions belonging to one region of search space, say, Ri, and all deletions belonging to another region, say, Rj, where $Ri \neq Rj$ and $Ri \cap Pj = \varnothing$. However, as the system proceeds, it learnt that Dyn is related to Deven. As a result, Dyn measurement needs to be refined. The learning process is this. From original case (0) to case (4), we have:

case	insertion	deletion	Deven	original Dyn	change 1	change 2
grid	[0] [1] [2] [3]	[0] [1] [2] [3]	25.0%			
0->1	0, 0, 2, 0	0, 1, 0, 2	33.3 %	21.25%	6.25%	31.25%
1->2	0, 1, 1, 1	2, 0, 0, 1	37.5 %	37.50%	0.00%	25.00%
2->3	1, 2, 3, 2	0, 0, 2, 2	37.5 %	75.00%	25.00%	25.00%
3->4	0, 4, 3, 3	1, 2, 0, 1	50 %	87.50%	37.50%	50.00%

As a learning process the system has knowledge of expected correlation between Dyn and Deven. This knowledge indicates that the dynamic factor Dyn directly influences data distribution Deven. It is very accurate to say that heuristically $\Delta Dyn \propto \Delta Deven$, where $\propto$ indicates a proportional relationship.

The system also knows that Deven is calculated on the resolution level of a search space and therefore, the accuracy will be at that level. When the system gets the results from each case, it sees no expected relation between Dyn and Deven, for the original Dyn given by the following calculation:

case	Deven	original Dyn
	25.0%	
0->1	33.3 %	21.25%
1->2	37.5 %	37.50%
2->3	37.5 %	75.00%
3->4	50 %	87.50%

When comparing ΔDyn and $\Delta Deven$ the system can detect that from case (2) to (3), $\Delta Deven = 0$ whereas $\Delta Dyn = 75\%$; which is double the value of the previous case. The reason is that Deven is not changed as the final effect over the Deven is the same compared to the previous case. Consequently, Dyn = Ir + Dr only partially reflects the dynamic change of the application. The system therefore, uses Dyn = |Ir - Dr| to calculate dynamic factor. The result becomes:

case	Deven	after change 1 Dyn
0->1	33.3%	6.25%
1->2	37.5 %	0.00%
2->3	37.5 %	25.00%

3->4	50 %	37.50%

After the first change, the system can again detect that from case (2) to (3), ΔDeven = 0 so that ΔDyn for case from (1) to (2) and from (2) to (3) should be the same. As Dyn for case (1) to (2) are 0% whereas ΔDeven = 4.2%, the system learnt that the inaccuracy is caused by calculating the dynamic factor over the entire search space. As mentioned before, Deven is measured at resolution level. To maintain the expected relationship between ΔDeven and ΔDyn, Dyn has to be calculated at the resolution level. As a result another change has been introduced for calculating Dyn, which is changed to:

$$Dyn = \sum_{i=0}^{i=r} | Ir[i] - Dr[i] |,$$

where Ir[i] and Dr[[i] are insertion and deletion rates for grid [i].

The result is:

case	Deven	after change 2 Dyn
	25.0%	
0->1	33.3 %	31.25%
1->2	37.5 %	25.00%
2->3	37.5 %	25.00%
3->4	45% %	50.00%

Now the expected relation is established. The system will use the formula for Dyn from change 2 to estimate the Dyn feature.

In this section several examples are given to show how the expert system refines its ability to reason more accurately. This validation is based on both experimental method and "learning by expected result" approach. The experimental method allows the system to modify and adjust its original knowledge based on applications, so that new knowledge can be incorporated with the knowledge system. The "learning by expected result" approach, enables the system to reason on its original knowledge by comparing the result, it gets from an application with

the expected result to refine knowledge.

4.9.6. Conclusion

No knowledge is perfect. "The world is not a fixed, solid array of objects, out there, for it cannot be fully separated from our perception of it. It shifts under our gaze, it interacts with us, and the knowledge that it yields has to be interpreted by us. There is no way of exchanging information that does not demand an act of judgement. " [BR73]. An expert system is not an exception in terms of perfection. It simply simulates an expert using programs. The computer has the advantage of speed, but it can misinterpret knowledge and therefore, a tuning and verification element in the system is a must.

Chapter 5 Conclusion

The problem addressed by this research effort was to apply heuristics in selecting and tuning m-d (multi-dimensional) algorithms for physical database design. Efforts in computer applications have been directed at finding various algorithms for efficient organisation and use in database design. However, most of these algorithms are application-oriented and technical and therefore, difficult to use to their best effectiveness. Choosing an algorithm for an application requires consideration of the characteristics of applications. Identifying an implementation algorithm which will be the near-optimal choice becomes a challenging problem to tackle.

Given several algorithms and an application to derive a better solution, a physical database designer may start by analysing the application characteristics, hardware and software environment, and the requirements; analysing the features of available algorithms before determining an optimal implementation algorithm. There are a variety of applications using database technology and many of them have similar characteristics which influence the choice of implementation algorithm. Hence the same algorithm may be applied, to those which are categorised as the same application type, successfully with high probability. On the other hand, given a group of application types (different types have different characteristics) and a number of algorithms, by analysing the strengths and weaknesses of each algorithm, a heuristic judgement can be applied to determine which is more appropriate for an application type. The choice of an implementation for a data set in a computer is a fuzzy area; it depends on the application domain - the data distribution and the way these data are used and updated. Thus the selection is made based on analysis of both the algorithms and the applications.

Appendix A1

The analysis of the inverted file partition and the grid file partition

For simplicity, we assume that the size of a data item is equal to the size of a bucket.

(1) Inverted file

(a) The model of the inverted file partition:

Data set Ds = { d1, d2, ..., dn}

Suppose kj1 for j = 1, 2, ..., n is a primary key set:

dj = (kj1, kj2, ..., kjm) for j = 1, 2, ..., n

The data set is stored in a sequence so that ki1 < kj1 if i < j.

For kji where i ≠ 1 (not a primary key) and j = 1, 2, ..., n there are (m - 1) key sets created for the search purpose:

Kj = { kij | (i = 1, 2, ..., x) and (j = 2, 3, ..., m) and (kij < klj if i < l) }

(b) The complexity of the model for the inverted file partition:

Storage complexity

The main data set requires storage of |Ds| = O(n).

The non-primary key sets.

Suppose the number of different values for non-primary keys are N2, N3, ..., Nm so that the average storage requirement Savg for Kj where j = 2, 3, ..., m, can be derived as:

$$Savg = \frac{N2 + N3 + \ldots + Nm}{(m - 1)}$$

In the formula, the complexity of Ni for i = 1, 2, ..., m is O(n).

The storage required for non-primary keys are:

(m - 1) x Savg = O(m) x O(n) = O(m x n) and

the total storage required for the data set is:

S = |Ds| + (m - 1) x Savg = O(n) + O(m x n) = O(m x n)

The complexity of the storage, therefore, is **O(n x m)**.

The time complexity

The time complexity is related to the geometric proximity. If the data items are stored closely together, in terms of a retrieval query requirements, then the time complexity can be reduced. If we view a query as a classification function which generates a result space, and the partition over the data set as a number of data regions, then the complexity is evaluated by two factors: (i) The access paths. (ii) How closely a result space matches the data region(s) by the partition over the data set, i.e. if the minimum number of data regions required to include the result space is small, then it is close a match. The time complexity, therefore, depends on both the result space generated by a query and the data region generated by the partition.

(2) Grid file partition

(a) The model of the grid file partition.

Data set Ds = { d1, d2, ..., dn}

dj = (kj1, kj2, ..., kjm) for j = 1, 2, ..., n

The data set is stored in a sequence so that di before dj if z(Ii) < z(Ij).

(b) The complexity of the model for the grid file partition.

Storage complexity

The main data set requires storage of |Ds| = O(n).

If the grid partition is implemented using an indexing approach then the complexity of the index will be O(n).

Therefore the complexity of the storage is **O(2 x n)**.

The time complexity

The time complexity is related to the geometric proximity which is the same as in the inverted file partition. It depends on the characteristics of an application.

(3) Comparison

For storage complexity when m > 2 the grid file partition outperforms the inverted partition. For time complexity, if most queries only involve the primary key, then the inverted file partition will be a better choice, otherwise the grid file partition will generate a smaller result space which involves fewer storage accesses.

Appendix A2

Application features calibration

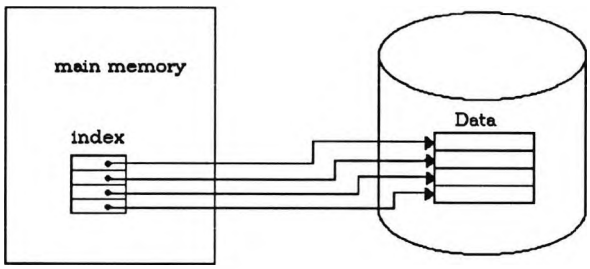
Deciding the limit of C11, C13, C21 and C31

The choice between an indexing or a hashing function to implement PT1 in terms of storage and speed is determined by the following estimation. For ease of discussion, we shall assume that available main memory for the data set is M, the number of total grid cells is 2^L (L is the level of the data set), the number of empty grid cells is Nempty, the index record length is Sidx, and the bucket size is b. Our discussion is based on the consideration of speed and storage utilisation.

(i) Speed

The speed is mainly influenced by the storage of access paths. There are two cases for the indexing implementation shown in Figure 1. (a) and (b) respectively. In case (a) the entire index can be stored in main memory so that we can get the address of the required data item(s) from the index file. Only one secondary storage access is required to get the data item. There is no difference between an indexing method and a hashing method. In case (b) there is not enough main memory for the entire index file and therefore, to get the address of a data item we have to read the index file into main memory first, then perform another read to get the required data item. An extra secondary storage access is needed to compare the indexing algorithm with a hashing algorithm. Thus for the indexing approach the speed depends on where the index file is stored.

(a) the index file can be stored in main memory



To retrieve a data item:

(1) get the address from the index file in main memory

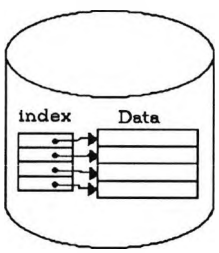
(2) access the data item on the secondary storage

T = Tmemory + Tsecondary ≈ Tsecondary

(b) the index file cannot be stored in main memory

Tmemory: time for main memory access

Tsecondary: one access to secondary storage



To retrieve a data item:

(1) get the address from the index on the secondary storage

(2) access the data item from the secondary storage

T = Tsecondary + Tmemory + Tsecondary ≈ 2 ×secondary

Figure 1. Indexing implementation.

The storage requirement for an index file can be expressed as **Sidx x 2^L**. When **Sidx x $2^L \leq$ M**, the entire index could be stored in main memory. In order to find a data item, an array-like calculation determines an entry to the index file. Based on the content of the index entry the data item can be located in secondary storage. When **Sidx x 2^L > M** the index has to be stored on the secondary storage and as a result, an extra access will be needed to get the required data item(s).

(ii) Storage utilisation

The storage utilisation is determined by the extra amount of space required to store access paths of a data set for retrievals. An indexing implementation needs extra space to store an index file. A hashing approach needs extra space to maintain regularity, which is a one-to-one mapping between a grid cell and a data bucket, to carry out a grid-cell-address (or a signature-address) calculation. Hence the extra space required for an index file is **Sidx x 2^L**, and a hash mapping is **Nempty x b**. If **((Nempty x b) > Sidx x 2^L) and (Sidx x $2^L \leq$ M)** then the index method is preferable. If speed is not an important factor then the condition can be relaxed to be **(Nempty x b) > Sidx x 2^L**, otherwise the hashing method is favourable.

C11 is decided on the basis of requirements. If speed is not important for an application then C11 can be determined by the storage. If speed is important then C11 can be decided by both the storage and speed. Hence the meaning of storage utilisation is determined by the requirement of an application and hardware/software limitations. Based on the above analysis we can derive the following conditions.

(1) Storage is the main factor for the requirement:

C11 = (Nempty x b)> Sidx x 2^L)

(2) Speed is the main factor for the requirement:

C11 = (Sid x 2^L) $\leq$ M) and (Nempty x b)> Sidx x 2^L)

We can also derive conditions:

C13 = (Sidx x 2^L) $\leq$ M)

C21 = (Sidx x 2^L) > M)

C31 = (Nempty / r > 50%)

Determine conditions of C14 and C25

The dynamic insertions can be described by the database state transformation. Let S^b represent the current state of the database and S^a represent the state after an insertion. We can define S^b as:

$S^b = \{Nempty^{(b)}, Nfull^{(b)}, Ncell^{(b)}\}$

A split results from adding a data item to the data space. It may change the system state.

(1) A data item is added to one of the empty grid cells

$S^b \text{ --> } S^a = \{ Nempty^{(b)} - 1, Nfull^{(b)}, Ncell^{(b)} + 1 \}$

When a new data item is added to an empty grid cell, a new data bucket will be allocated to the data set by an indexing implementation. An index entry needs to be updated and its address will refer to a new bucket.

(2) A data item is added to one of the non-full grid cells

(a) $S^b \text{ --> } S^a = S^b$ OR

(b) $S^b \text{ --> } S^a = \{ Nempty^{(b)}, Nfull^{(b)} + 1, Ncell^{(b)}\}$

In the above two cases the insertion does not cause a split and the time required to insert an item is equal to the sum of the block access time and a block write time.

(3) A data item is added to one of the full grid cells, resulting in a split:

$S^b \text{ --> } S^a = \{ Nempty^{(b)}, Nfull^{(b)} - 1, Ncell^{(b)} + 1\}$, for an indexing method OR

$S^b \text{ --> } S^a = \{ Nempty^{(b)}, Nfull^{(b)} - 1, 2 \text{ x } Ncell^{(b)} \}$ for a hashing method

Here the number of grid cells added from the split is equal to $Ncell^{(b)}$ and the level of the data set L has been increased by 1, i.e. $L(a) = L^{(b)} + 1$ where $L^{(b)}$ indicates the data set level before an insertion and $L^{(a)}$ is the data set level after an insertion.

Using an index file the number of index entries will be doubled and the contents of

the index need to be rearranged in order to be located by $L^{(a)}$. This process includes setting pointers and ordering the index records in the z-order. This reflects the complexity of the insertion. The C11 condition will be re-examined as the index size has been doubled; but in physical data space (physical data space refers to the space in secondary storage which corresponds to the partitioned data space) only the grid cell that causes the split needs to be reorganised. Adding one data item will cause these data items, in one data bucket at most to be reorganised and one more new bucket to be allocated in the physical data space.

Using a hashing function the number of data buckets requiring reorganisation will be equal to the number of data cells before the split. It has the same complexity as reorganising the index, but instead of an index, the reorganisation takes place within the data file itself. There is a requirement to order the data buckets in z-order.

For the S^b state, the probability of a data item being added to the full bucket is $Nfull^{(b)}/Ncell^{(b)}$. In cases (1) and (2) it does not cause a split, so the effort involved is insignificant. When a split occurs, which is the case in (3), we need to choose between available implementation algorithms.

$$C14 = ((N^{(a)} \times Sidx) < M) \text{ and } (Dyn \times \frac{N^{(b)\,full}}{N^{(b)\,cell}} \leq 30\%))$$

The condition C14 is determined by heuristics. Heuristics that make a decision on the basis that the indexing method copes with a dynamic situations better than the z-hashing and the PLOP-hashing is better than the indexing approach. The decision as to which approach should be applied is difficult to make. In construction of heuristic function we have assigned the value by a comparison with other algorithms, that is by comparison with other algorithms to see where a particular algorithm stands.

A range search in a m-d data space retrieves a group of data items which satisfy a certain condition. These conditions quantify a geometric closure enclosing those required data items. It can be described as " search all data items that fall within the boundaries of $[a_{i1}, a_{i2}]$ for i = 1, 2, ..., m ". A partial range search means that if T

$= a_1 a_2 \ldots a_m$ represents a tuple and the query is in the form of $Q = ??a_x a_{(x+1)}??$, implying that if only the x^{th} and $(x+1)^{th}$ attributes are concerned during a search then Q is a partial range query. Alternatively, it can be described as " search all data items within a set of boundaries of $[a_{i1}, a_{i2}]$ for i = 1, 2, ..., k and k < m ". The data items that satisfy the range or partial range search usually result in accesses to several grid cells, that is, several data buckets need to be consulted in order to get the required data items. To improve the efficiency of a search it is desirable to arrange data items in secondary storage according to the geometric distance between data items. The z-hashing provides a better geometric proximity for organising data on secondary storage than the EXCELL algorithm. This is due to the fact that the EXCELL algorithm keeps the addresses in the index file, where the index records are ordered in z-order; but it does not necessarily guarantee that the data items are also held in z-order. When a range search is required frequently in an application, the z-hashing improves the performance as a whole by accessing the data set in physically consecutive buckets. To quantify the level of frequency for the range and partial range searches information about their frequencies needs to be recorded. In the section " dynamic changes of database profile " we have introduced $Rs^{(d)}$ as the range search rate. This can be evaluated by counting the number of range and partial range searches conducted over a data set. The initial value Rs can also be estimated by users and supplied from the USI. Hence, the high range search rate can be defined as:

$$C25 = Rs > 50\%$$

To identify the different effects between various implementation algorithms we can analyse their likely performance for range and partial range searches. For the EXCELL algorithm, as analysed above the performance for range searches cannot be guaranteed. For the PLOP-hashing algorithm, which copes with the dynamic situation well, the geometrical proximity can be lost by numbering the slices according to the growth order.

Deciding the condition of C24, C33 and C42

The expected insertion pattern depends on the current state of a data set. A split will increase the data set level L by 1 and consequently it will change the resolution of the data space and create a set of unused bucket sequence numbers $[2^L, 2^L + 1, \ldots, 2^{L+1}]$. To match the order set by the split rule the order of insertions can be worked out in terms of which sequence of the grid cells should be split, i.e. the

ranges given by these grid cells form a sequence of data items expecting to be inserted. Suppose the sequence of the grid cells is $(G_1, G_2, ..., G_x) \in (0, 1, ..., 2^L - 1)$, the range of each grid cell is $R[G_i]$ for i = 1, 2, ..., x, and data items to be added are $(d_1, d_2, ..., d_y)$. Firstly we transform these data items into their equivalent z-codes: $d = (d_1, d_2, ..., d_y) \rightarrow Z = (z(d_1), z(d_2), ..., z(d_y))$ belonging to $(0, 1, ..., 2^L - 1)$. Secondly we order $Z = (z(d_1), z(d_2), ..., z(d_y)) \rightarrow Z' = (z'(d_1), z'(d_2), ..., z'(d_y))$ corresponding to$(G_1, G_2, ..., G_x)$ where $z'(di) \in G_i$ and z'(d(i+1)) belongs to Gk and i < k. Elements in Z' represents the insert sequence. The insertion pattern can be expressed as:

C24 = ((the z-code for the elements in the insertion set matches the order of Z') or (Dyn < 10%))

C32 = (Dyn < 10%)

C42 = (Dyn > 50%)

Determine C23

Even data distribution (EDD) measurement.

Suppose that the number of data divided into m grid cells and the number of data items in each cell is stored in C[i] for i = 1, 2, 3, ..., r.

The EDD meets the condition for packing density p,

$$C23 = \sum_{i=1}^{m} |C[i] - b| < (1 - p) \times r \times b$$

Appendix A3

An explicit illustration of the split rule

Suppose a split is a state transformation $S^{(b)} \dashrightarrow S^{(a)}$. From data space level L to (L + 1), the state can be described by bucket sequence numbers as:

$S^{(b)} = (0, 1, 2, ..., 2^L - 1)$

$S^{(a)} = (0, 1, 2, ..., 2^L - 1, 2^L, 2^L + 1, ..., 2^{(L+1)} - 1)$

The first grid cell to be split will be the one which produces the number of 2^L, $2^L + 1$. Assuming that the grid cell k within the range of $S^{(b)}$ is the chosen grid cell then the next grid cells to be split will be the one which produces the z-codes $2^L + 2$ and $2^L + 3$. This process can then be described as a recursive process, until a full split is reached.

First split:

Choose x which produces numbers among the available bucket numbers 2^L to $2^L + 1$ so that the produced numbers are the minimum number among the available unused numbers.

Second split:

Choose y which produces bucket numbers $2^L + 2$, $2^L + 3$.

Third split:

If x, y can be derived from a bucket number in use, i.e. the grid cell z, which belongs to the used z-number, can produce x, y then we choose z to split, otherwise we choose the one which produces bucket number $2^L + 4$, and $2^L + 5$ and so forth. An example which illustrates the process is given in Figure 2.

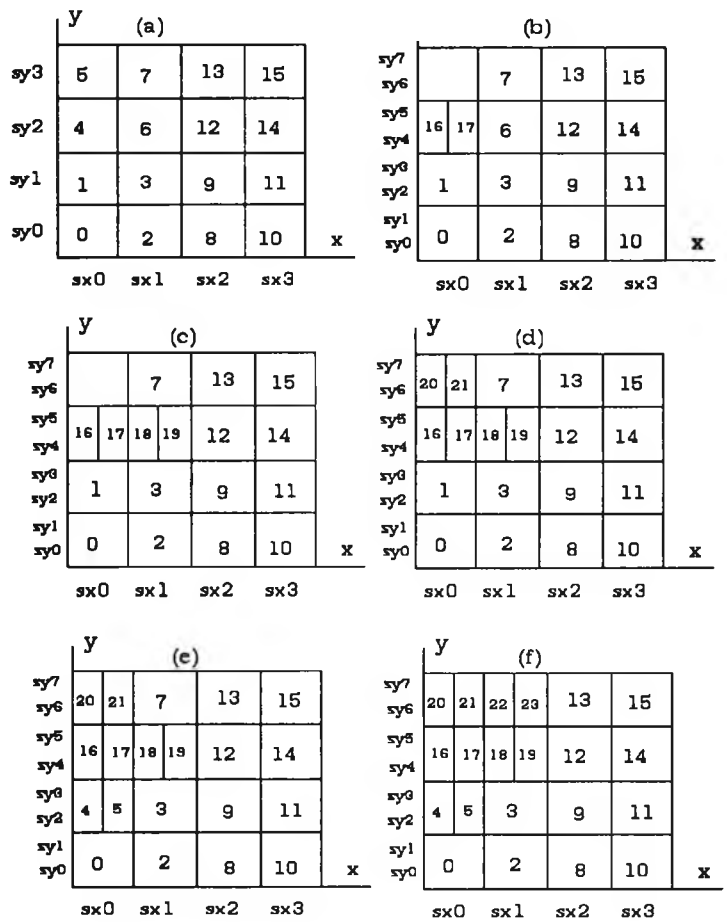


Figure 2. z-hashing splitting sequence.

Appendix A4

Bang file and z-hashing storage utilisation

For the z-hashing the storage utilisation depends on the data distribution and is measured by the packing density for each grid cell. If the number of data items in a grid cell is C(z) for z = 1, 2, ..., Ncell and the bucket size is b then the average storage utilisation Su for the data set will be:

$$Su = \frac{\sum_{i=1}^{i=Ncell} C[i]}{b \times Ncell}$$

The extra space, |additional space|, required for the data set, therefore, is

$$|\text{additional space}| = (1 - Su) \times Ncell$$

If the index record length is Ridx then the storage required for the index file, |index|, by Bang partition will approximately be:

$$|\text{index}| = Ridx \times \lceil n / b \rceil \times Su$$

where n is the number of data items in the data set.
Therefore, we can derive the formula for the break-even point calculation as:

$$|\text{additional space}| - |\text{index}| < 0$$

If the condition holds the z-hashing is preferable otherwise the Bang file partition is favourable.

Appendix A5

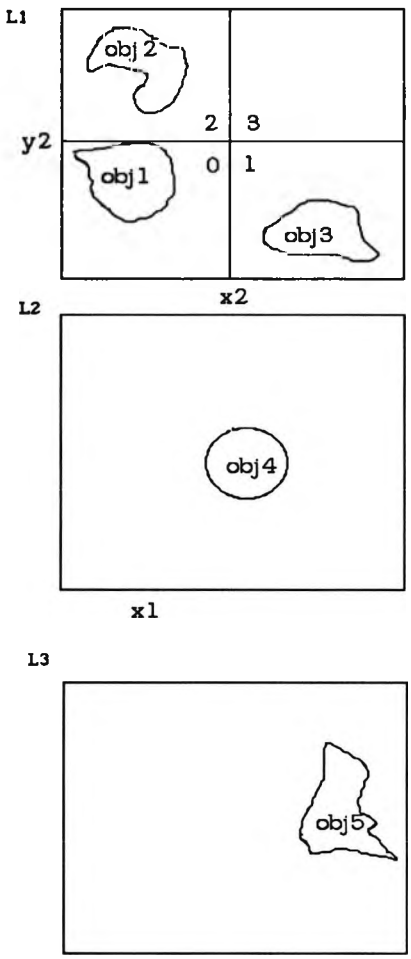
Calculation for a multi-layered m-d identifier

Let G = {G1, G2, ..., Gk} be k-layers of a grid partition for an object data set. Let obj = {objx1, objx2, ..., objxk} be the entire object data set and objxi for i = 1, 2, ..., k refer to a subset of objects in layer i, i.e. objxi = {obji1, obji2, ..., objixi}. For each layer the element is represented by oblxi = {idij, Wij for i = 1, 2, ..., k; j = 1, 2, ..., xi}, where idij is the object identifier formed by z-code and Wij is a weight function which decides the number of buckets each object occupies. Initially Wij = 0. A total number of buckets (Ni for i = 1, 2, ..., k) used for each layer is kept for easier formation of the identifier. An identifier for an object in layer i can be calculated by:

$$idij = \sum_{x=1}^{x=(i-1)} Nx + \sum_{y=1}^{y=(j-1)} Wiy$$

Wiy = number of buckets required to store the object.

Having used this method to calculate the object identifier we noticed that when variable length objects are allowed an insertion may cause changes to these identifiers, implying high complexity of change (n/2 = O(n) in average). Hence it cannot cope with dynamic situations efficiently. Alternatively, we can keep the address of an object stored in the index without changing the object identification. This means that objects stored in different layers can have the same identifier distinguished by their layer and the address. An object is identified by its minimal enclosure in a partition for a layer. This is shown in Figure 3.



object	weight	layer
obj1	w11 = 2	1
obj2	w12 = 2	1
obj3	w13 = 4	1
obj4	w21 = 3	2
obj5	w31 = 4	3

$$idij = \sum_{x=1}^{x=i-1} Nx + \sum_{y=1}^{y=j-1} wiy$$

where idij is the object identifier for object j in layer i. Nx is the number of buckets occupied in layer x. wij is weight given for a correspond object.

$$id11 = \sum_{x=1}^{x=0} Nx + \sum_{y=1}^{y=0} w1y = 0$$

$$id12 = \sum_{x=1}^{x=0} Nx + \sum_{y=1}^{y=1} w1y = 2$$

$$id13 = \sum_{x=1}^{x=0} Nx + \sum_{y=1}^{y=2} w1y$$

$$= w11 + w12 = 4$$

$$N1 = w11 + w12 + w13 = 8$$

$$id21 = \sum_{x=1}^{x=1} Nx + \sum_{y=1}^{y=0} w2y$$

$$= w11 + w12 + w13 = 8$$

$$N2 = N1 + w21 = 11$$

$$id31 = \sum_{x=1}^{x=2} Nx + \sum_{y=1}^{y=0} w2y = 11$$

Figure 3. Object identifier calculation for a multi-layered grid partition.

Appendix A6

Performance evaluations for different applications

(1) The EXCELL algorithm

ACCESS TIME ESTIMATION

(a) Index file is stored in main memory

POINT DATA ACCESS

T = Tsec

T - time estimated for a point search.

Tsec - time required for an access to secondary storage.

RANGE SEARCH

$$a = \frac{avg(C[i]) \times k}{b \times k} = \frac{avg(C[i])}{b}$$

a - an estimation of access accuracy of a range search.

avg(C[i]) - the average number of data items in a grid cell:

avg(C[i]) = n / B

where B is the number of buckets actually in use.

Let Dreq = ([Dx1, Dx2], [Dy1, Dy2]) be the range covered by the query, Ix be the interval length in the x dimension, Iy be the interval length in the y direction. We can calculate the number of blocks require to be retrieved.

b - bucket size.

k - number of grid cells a range search covered.

$$k = \left\lceil \frac{Dx2 - Dx1}{Ix} \right\rceil + \left\lceil \frac{Dy2 - Dy1}{Iy} \right\rceil$$

INSERT A DATA ITEM

T = one read + one write = 2 x Tsec when there is no split

T = one read + two writes = 3 x Tsec if there is a split

To insert a data item the maximum time required is when an insertion causes a split; thus one data block needs to be read and split into two data blocks. Subsequently, these two blocks need to be written to the secondary storage, requiring two write operations. Moreover, the complexity of reorganising the index file is O(r), r is the current file resolution.

DELETE A DATA ITEM

(i) no merge

T = one read + one write = 2 x Tsec

(ii) there is a merge

T = two reads + one write = 3 x Tsec

(b) Index file cannot be stored in main memory

POINT DATA ACCESS

T = 2 x Tsec

one access to the index file and one for the data block.

RANGE SEARCH

$$a = \frac{avg(C[i]) \times k}{1.5 \times b \times k} = \frac{avg(C[i])}{1.5 \times b}$$

k - number of grid cells a range search covered. The factor 1.5 indicates the average index accessed is 0.5 of the total number of secondary storage access required.

INSERT A DATA ITEM

T = two reads + one write = 3 x Tsec when there is a split

T = two reads + two reads when there is a split

A read to the index is added to the previous case in (a).

DELETE A DATA ITEM

(i) there is no merge

T = 2 x reads + one write = 3 x Tsec

(ii) there is a merge

T = three reads + one write = 4 x Tsec

(2) the z-hashing algorithm

ACCESS TIME ESTIMATION

(a) With overflow handling

POINT DATA ACCESS

With overflow handling to the z-hashing method an access to a data item depends on two factors.

(i) Whether the data item to be accessed is in the home area or in the overflow area.

(ii) The technique used to handle the overflow. In our system the overflow is handled by a separator table. The details are in the Appendix A8.

For overflow handled by the separator table we assume that the table can be stored in main memory otherwise a split is triggered to resolve the table. Under this assumption we get:

$$T = Tsec$$

A data item to be accessed by a query will relate to a specific grid cell represented by its z-code. The separator (overflow) table in main memory will be searched and there are two situations:

(i) The z-code is in the table. This implies that the required data item may be in the overflow bucket so that the separator is compared with the relevant attribute value in the query. It can be determined that either the data item is stored in home or in the overflow areas and therefore, a secondary storage access is required to read the block into main memory for examination.

(ii) The z-code is not in the table. One secondary storage access is needed to read the block which may contain the data item.

RANGE SEARCH

$$a = \frac{avg(C[i]) \times k}{b \times k'}$$

where

k - the number of grid cells covered by the range query

k' - the number of data blocks needed to be accessed by the range search. There are two cases:

(i) Every grid cell covered by the query is in a home grid so that k = k'.

(ii) Some of the grid cells covered by the query need overflow areas to store data items falling in the range specified by the query. As data items for a range search domain are beyond the storage capacity for a grid cell, both home and the overflow blocks have to be accessed to get these required data items, and therefore k' > k. The exact value of k' depends on the number of grid cells which demand overflow areas and the depth of an overflow area. It can be estimated by the following formula.

$$k' = \sum_{i \in DRs} \left\lceil \frac{C[i]}{b} \right\rceil$$

where DRs is the domain of the range search, i.e.
DRs = (Dx1, Dx2) X (Dy1, Dy2)

INSERT A DATA ITEM

There are several cases to be dealt with when inserting a data item.

(i) insert a data item into a non-full grid cell

T = one read + one write = 2 x Tsec

(ii) an insertion results in an overflow

T = one read + two writes = 3 x Tsec

(iii) A split is triggered by an insertion

When a split is triggered the resolution of the data set changes and the file needs a reorganisation

T $= r^{(b)}$ reads + $r^{(a)}$writes = $(r^{(b)} + r^{(a)})$ x Tsec

$= 3 \times r^{(b)} \times Tsec = 3 \times r \times Tsec$

where r is the simplified form of $r^{(b)}$.

$r^{(b)}$ - resolution before a split

$r^{(a)}$ - resolution after a split $r^{(a)} = 2 \times r^{(b)}$

DELETE A DATA ITEM

(i) a deletion empties the overflow bucket

T = one read = Tsec

and a deletion is needed for the separator table,

(ii) a deletion empties a home bucket

there is an overflow area corresponding to the home block

T = two reads + one write = 3 x Tsec

where

one read to home bucket

one read to the overflow bucket

one write to the home bucket

otherwise

T = one read = Tsec

and an insertion to the empty grid cell table,

(iii) a deletion is one of the data items in the home or overflow bucket

T = one read + one write = 2 x Tsec,

(iv) a deletion causes a merge operation

In this situation a file reorganisation is triggered.

$T = (r^{(b)} + r^{(a)}) \times Tsec = 1.5 \times r \times Tsec$

where $r^{(a)} = 1/2 \times r^{(b)}$.

(b) Without overflow handling

POINT DATA ACCESS

T = Tsec

A given data item uniquely corresponds to a grid cell and a grid cell uniquely corresponds to a data bucket, and thus only ONE disk access is required for a point data access.

RANGE SEARCH

$$a = \frac{avg(C[i]) \times k}{b \times k} = \frac{avg(C[i])}{b}$$

$$k = \left\lceil \frac{Dx2 - Dx1}{Ix} \right\rceil + \left\lceil \frac{Dy2 - Dy1}{Iy} \right\rceil$$

k - the number of data blocks covered by the range search, i.e. the number of data blocks needed to be retrieved.

n(req) = avg(C[i]) x k

n(req) - an estimation of the number of data items required.

INSERT A DATA ITEM

T = one read + one write = 2 x Tsec if there is no split

T = 3 x $r^{(b)}$ x Tsec = 3 x r x Tsec if there is a split

where $r^{(b)}$ is the resolution before a split.

When there is a split the whole data file needs to be reorganised in order to establish a one to one relationship between the grid cell in the new resolution and the block number. We can see that a large amount of I/O accesses are required and so the z-hashing cannot cope with dynamic situations efficiently.

DELETE A DATA ITEM

T = one read + one write = 2 x Tsec if there is no merge

T = $r^{(b)}$ reads + 0.5 $r^{(b)}$ writes = 1.5 x $r^{(b)}$x Tsec if there is a merge

(3) The quantile-hashing algorithm

POINT DATA ACCESS

T = Tsec

RANGE SEARCH

$$a = \frac{n_i}{b \times k}$$

k - number of grid cells a range search covered.

n_i - number of data items searched.

INSERT A DATA ITEM

T = one read + one write

= 2 x Tsec when there is no split

T = $r^{(b)}$ reads + $r^{(b)}$ write

= ($r^{(b)}$ + ($r^{(b)}$ + sx(b))) x Tsec

= (2 x r + sx) x Tsec

(if there is a split in the y dimension)

T = $r^{(b)}$ reads + $r^{(b)}$ writes

= ($r^{(b)}$ + ($r^{(b)}$ + sy(b))) x Tsec

= (2 x r + sy) x Tsec

(if there is a split in the x dimension)

DELETE A DATA ITEM

(i) A deletion does not cause a merge operation

T = one read + one write = 2 x Tsec

(ii) A deletion causes a merge

T = $r^{(b)}$ reads + $r^{(b)}$ writes

= ($r^{(b)}$ + ($r^{(b)}$ - $sx^{(b)}$)) x Tsec

= (2 x r - sx) x Tsec

(if there is a merge in the y dimension)

T = $r^{(b)}$ reads + $r^{(b)}$ write

= ($r^{(b)}$ + ($r^{(b)}$ - $sy^{(b)}$)) x Tsec

= (2 x r - sy) x Tsec

(if there is a merge in the x dimension).

(4) The PLOP-hashing

POINT DATA ACCESS

T = Tsec

RANGE SEARCH

$$a = \frac{n_i}{b \times k}$$

$$n_i = \sum_{Ri \in RxXRy} avg(C[i])$$

$$Ri = \begin{cases} 0 & \text{if the region is not in RxXRy} \\ 1 & \text{if the region is in RxXRy} \end{cases}$$

k - number of regions covered by the range search.

INSERT A DATA ITEM

T = one read + one write = 2 x Tsec when there is no split

T = sx reads + 2 x sx writes = 3 x sx x Tsec if there is a split in the y dimension

T = sy reads + 2 x sy writes = 3 x sy Tsec if there is a split in the x dimension

DELETE A DATA ITEM

(i) a deletion does not cause a merge operation

T = one read + one write = 2 x Tsec,

(ii) a deletion causes a merge operation in the y dimension

T = sx reads + 1/2 sx writes = 1.5 x sx x Tsec

similarly in the x dimension, T = 1.5 x sy x Tsec.

(5) The BANG file

POINT DATA ACCESS

T = Tsec.

RANGE SEARCH

$$a = \frac{avg(C[i]) \times k}{b \times k} = \frac{avg(C[i])}{b}$$

k - the number of regions covered by the range search.

INSERT A DATA ITEM

T = one read + one write = 2 x Tsec when there is no split

T = one read and two writes = 3 x Tsec if there is a split

DELETE A DATA ITEM

(i) a deletion does not cause a merge operation

T = one read + one write = 2 x Tsec

(ii) a deletion causes a merge operation in the y dimension

T = two reads + one write = 3 x Tsec

similarly in the x dimension, T = 1.5 x sy x Tsec

STORAGE ESTIMATION

(1) EXCELL algorithm

$S = |index| + r \times b$

where

$|index| = Sidx \times r = |address| \times r$

(2) Z-hashing

(a) With overflow handling

$S = (r + Nover) \times b$

(b) Without overflow handling

$$S = 2^{\lceil \log_2 N(d) \rceil} \times r \times b$$

N(d) is the depth of the overflow.

(3) Quantile-hashing

$S = |index| + sx \times sy \times b$

where

$|index| = Sidx \times \sqrt{r}$

$= (|separator| + |address|) \times \sqrt{r}$

(4) PLOP-hashing

$S = |index| + sx \times sy \times b$

where

$|index| = Sidx \times \sqrt{r}$

$= (|separator| + |address| + |index|) \times \sqrt{r}$)

(5) BANG file

$S = |index| + \sqrt{r} \times b$

where

$|index| = Sidx \times \sqrt{r}$

$= (|level\ no| + |region\ no| + |address|) \times \sqrt{r}$)

Appendix A7

Information about distribution

the grid cell set which has more than b data items

$C(over) = \{ C[zi] \mid (C[zi] > b) \text{ for } i = 1, 2, ..., x \}$

the number of grid cells with more than b data item

$Nover = |C(over)|$

the depth of the overflow

$$N(d) = \left\lfloor \frac{\max (C[zi] \mid C[zi] \in C(over))}{b} \right\rfloor$$

the grid cell set which has no data items

$C(empty) = \{ C[zi] \mid (C[zi] = 0) \text{ for } i = 1, 2, ..., y \}$

the number of grid cells without any data items

$Nempty = |C(empty)|$

the storage utilisation estimation (or storage utilisation)

$$p = \frac{Nmin}{Nmax}$$

the minimum number of buckets required for a data set.

$Nmin = r$

the maximum number of buckets required for a data set

$$Nmax = 2^{\lceil \log N(d) \rceil} \times r$$

the degree of even data distribution

$$d(even) = \frac{\sum_{i=1}^{i=r} |C[i] - b|}{r \times b}$$

alternatively we can express d(even) as:

$$d(even) = \frac{\sum_{o=1}^{o=x} 2 \times (C[o] - b)}{r \times b} \qquad \text{<i>}$$

when $(n - \lfloor n / b \rfloor) = 0$, here $C[o] \in C(over)$ and

$$C(over) = \{ C[o] \mid C[o] > b \text{ and } o = 1, 2, ..., x \}.$$

Let us prove the above expression <i>.

known (a) $n = \sum_{i=1}^{i=r} b$, (b) $\sum_{i=1}^{i=r} C[i] = n$, and

proving

$$\sum_{i=1}^{i=r} |C[i] - b| = \sum_{o=1}^{o=x} 2 \times (C[o] - b) \qquad \text{<ii>}$$

PROOF

from (a) and (b): $\sum_{i=1}^{i=r} (C[i] - b) = 0$

dividing C[i] for i = 1, ..., r into two groups C1[i] for i = x1, ..., xi and C2[j] for j = y1, ..., yj where all C1[i] > b, all C2[j] < b and $| \{x1, ..., xi, y1, ..., yj\}| = r$.

Let us look at:

$$\sum_{i=1}^{i=r} |C[i] - b| \qquad \text{<iii>}$$

representing C1[i] and C2[j] in the following manner:

C1[i] = c1[i] + b, C2[j] = b - c2[j]

according to the definition of C1[i] and C2[j] that c1[i] and c2[j] are greater than zero.We need to prove:

$$\sum_{i=1}^{i=r} |C[i] - b| = \sum_{j=y1}^{j=yj} 2 \times (C2[i] - b) \qquad \text{<iv>}$$

The formula <iii> can be represented as:

$$\sum_{i=1}^{i=r} |C[i] - b| = \sum_{i=x1}^{i=xi} (C1[i] - b) + \sum_{j=y1}^{j=yj} (b - C2[j])$$

$$= \sum_{i=x1}^{i=xi} (c1[i] + b - b) + \sum_{j=y1}^{j=yj} (b - (b - c2[j]))$$

$$= \sum_{i=x1}^{i=xi} c1[i] + \sum_{j=y1}^{j=yj} c2[j] \qquad \text{<c>}$$

the formula can also be represented as:

$$\sum_{i=1}^{i=r} |C[i] - b| = \sum_{i=x1}^{i=xi} (C1[i] - b) + \sum_{j=y1}^{j=yj} (b - C2[j])$$

$$= \sum_{i=x1}^{i=xi} (C1[i] - b) + \sum_{j=y1}^{j=yj} (b - C2[j])$$

$$+ \sum_{j=y1}^{j=yj} (C2[j] - b) - \sum_{j=y1}^{j=yj} (C2[j] - b)$$

$$= \left[\sum_{i=x1}^{i=xi} (C1[i] - b) + \sum_{j=y1}^{j=yj} (C2[j] - b)\right]$$

$$+ \sum_{j=y1}^{j=yj} (b - C2[j]) - \sum_{j=y1}^{j=yj} (C2[j] - b)$$

$$= \sum_{j=y1}^{j=yj} ([0] + (b - C2[j]) - \sum_{j=y1}^{j=yj} (C2[j] - b)$$

$$= \sum_{j=y1}^{j=yj} (b - C2[j] - C2[j] + b)$$

$$= \sum_{j=y1}^{j=yj} 2 \times (b - C2[j])$$

$$= \sum_{j=y1}^{j=yj} 2 \times (b - (b - c2[j]))$$

$$= 2 \times \sum_{j=y1}^{j=yj} c2[j] \qquad \text{<d>}$$

from <c> and <d> we know that:

$$\sum_{i=x1}^{i=xi} c1[i] = \sum_{j=y1}^{j=yj} c2[j]$$

that is, we can present formula <c> as:

$$\sum_{i=1}^{i=r} |C[i] - b| = \sum_{i=x1}^{i=xi} (C1[i] - b) + \sum_{j=y1}^{j=yj} (b - C2[j])$$

$$= \sum_{i=x1}^{i=xi} (c1[i] + b - b) + \sum_{j=y1}^{j=yj} (b - (b - c2[j]))$$

$$= \sum_{i=x1}^{i=xi} c1[i] + \sum_{j=y1}^{j=yj} c2[j]$$

$$= \sum_{i=x1}^{i=xi} 2 \times c1[i]$$

$$= \sum_{i=x1}^{i=xi} 2 \times (C1[i] - b)$$

therefore <iv> is proven.

When d(even) < x% the upper bound for the number of the empty grid cells will be x% x r. This is easily proved by the definition of d(even).

As far as the z-order is concerned four grid cells make up a combination to extract the features from a data space. Hence when a resolution r is chosen, information on a resolution with $r / (2^i)$ where i = 1, 2, ..., $\log_2 r - 2$, is easily derived. If the resolution is r/2 then the grid cell will grow to twice the size as for resolution r. When at resolution r, sx = 2 x sy is the case, we assume that every two slices in the x direction will be merged to one, and similarly if sy = 2 x sx, we assume that every two slices in the y direction will be merged to one. When sx = sy we can choose any one of them to merge. Suppose the resolution set is R = {r0, r1, ..., rx} where r0 = r, r1 = r/2, ..., rx = r / ($\log_2 r$). The rule given for d(even) is represented by a set of upper or lower bounds: d(even) = { < 20%, < 25%, < 50%, >50% } and the rule implies: (a) the comparison order is to be taken from left to right upon the set of upper bounds given; (b) the reason behind the values given is that the upper bound indicates the maximum number of empty grid cells for the data set. If an application has close upper bounds, say < 25% for all d(even) values of different resolutions, then the accuracy of estimation of even distribution to be bounded by 25% increases. The result is easily represented in the knowledge base by a bit matrix of (x) x (y), where x is the number of resolution levels and y is the number of upper bounds given in the rule. A rule as given above, has 4 upper bounds, and when there are also four levels, of resolution to be considered in an AAP, the following bit matrix is explained.

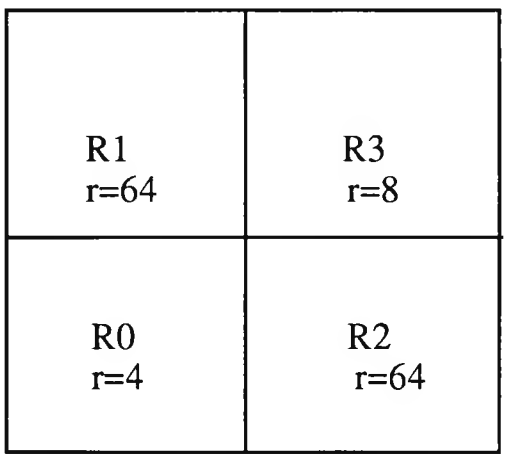
matrix		explanation
level r0	1 0 0 0	at r0 d(even) < 20%
level r1	0 1 0 0	at r1 d(even) < 25%
level r2	0 1 0 0	at r2 d(even) < 25%
level r3	0 0 1 0	at r3 d(even) < 50%

For an AAP the information on d(even) once derived can thus be stored as a bit matrix for system to use. Information gained at different resolution levels, such as C[i] for i = 0, 1, 2, 3 (the lowest resolution level), can also be used to tune the physical organisation under the system's control.

An illustration of using C[i] at resolution level of r = 4

By analysing information at different resolution levels a very small sized index file may be introduced into the system to allow varied resolution levels for different regions within the data space. An example is illustrated below.

Data space r = 4



It is possible that among these four regions {R0, R1, R2, R3} in the data space that R0 and R3 have sparsely populated data items, and R1 and R2 have a densely populated data distribution. To tune the physical organisation we may wish to apply different resolutions to differently populated regions. A small sized index thus can be used for this purpose. In each entry the resolution and the starting address is recorded for calculating an address by a z-hashing function. The tuning process is especially useful for a large data set with strong correlated data. An illustration is shown in Figure 4. Each region is treated as if it is an independent data set using the z-hashing implementation. We can see from Figure 4 that by introducing the index file, the reorganisation caused by a split may also be localised.

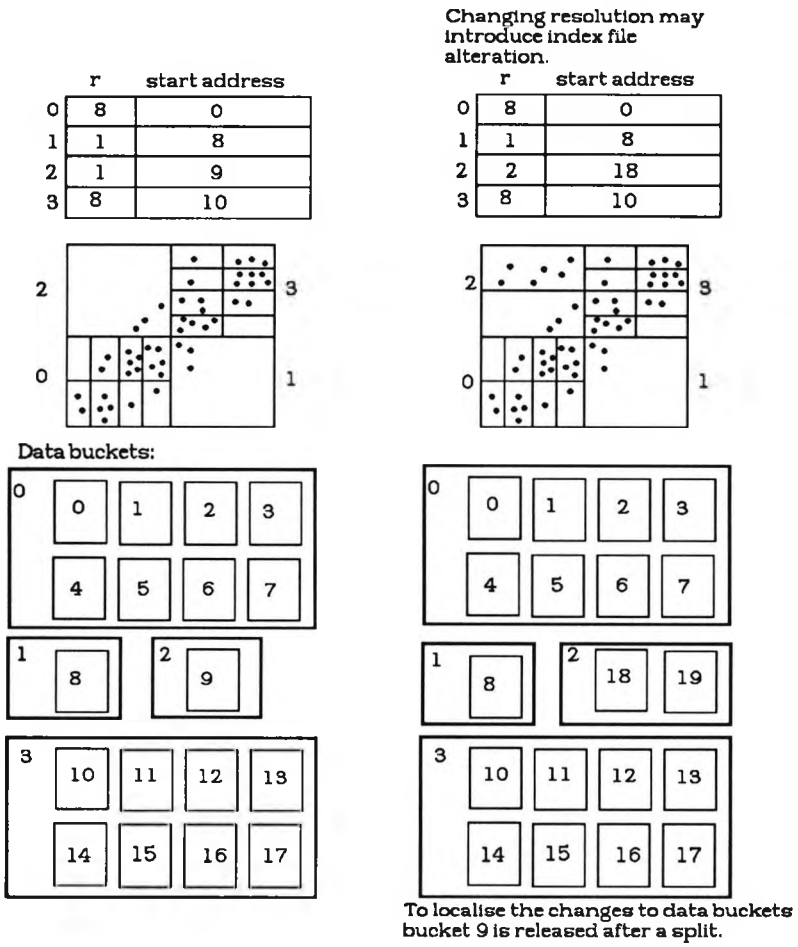


Figure 4. Tuning by different resolution.

The local data density Ld(sii)

The algorithm for calculating local data density

INPUT

C[i] for i = 0, 1, ..., r - 1

OUTPUT

Ld(xi) for i = 0, 1, ..., (sx - 1) $= \lfloor \log_2 r \rfloor - 1$

Ld(yj) for j = 0, 1, ..., (sy -1) $= \lfloor \log_2 r \rfloor - 1$

ALGORITHM

```
LD( )
{
        initialise Ld[xi] and Ld[yi] to be 0;
        for (y = 0; y++; y < sy)
        {
                for (x = 0; x < sx; x++)
                {
                z = interleaving x and y;
                Ld[y] = Ld[y] + C[z];
                }
        }
        for (x = 0; x++; x < sx)
        {
                for (y = 0; y < sy; y++)
                {
                z = interleaving x and y;
                Ld[x] = Ld[x] + C[z];
                }
        }
        for (x = 0; x++; x < sx)
                store Ld(x);
        for (y = 0; y++; y < sy)
                store Ld(y);
}
```

dynamic factor: $Dyn = \sum_{i=0}^{i=r-1} Ir[i] - Dr[i]$

access mode: random access or no special requirement

depth of overflow grid cells:

$$N(d) = \max\left(\left\lceil \frac{(C(o) - b)}{b} \right\rceil \right) \quad \text{for } C(o) \in C(over)$$

packing density:

$$p = \frac{N(d)}{\lceil \mathrm{Log}_2 N(d) \rceil}$$

Appendix A8

Overflow Handling

Overflow usually needs an extra access to the secondary storage device. The main purpose of applying a hash algorithm is to gain fast response. Naturally it is desirable to handle overflow without an extra access to the secondary device. In our system we handle the overflow by utilising an overflow table in the main memory for a hashing algorithm. When inserting a data item the grid cell which causes an overflow will be divided into two. The original z-code of the grid cell will be kept in the overflow table along with the dividing dimension chosen. A retrieval will always look up the overflow table to decide if the required data item is in the home or in the overflow bucket. The key to the method is to calculate the z-value for a data item, which is to be stored in as the first one in the overflow bucket, the separator.

The overflow/separator table

z-code	separator	dimension	layer	address

z-code - corresponding to the grid cell at current level which has caused an overflow.

separator - recording the position where the home and overflow area boundary value is in a specified dimension.

dimension - indicating the direction in which the boundary has been recorded as a separator.

layer - a grid cell can have more than 2 x b data items, so that within an overflow, another overflow may be embedded.

address - the address of the overflow area. To improve z-hashing storage utilisation the address can be a z-code for an empty grid cell. Using empty grid cells to accommodate overflow all z-codes for empty grid cells need to be recorded as free storage for the system to use.

An example

Suppose the grid cell x is a home grid cell which causes an overflow. In the current

grid cell x there are k items I = {i1, i2, ..., ik} where (k > b) have the same z-code x. An overflow process will split the set I into two sets: I1 = {i1, ..., im-1}, I2 = {im, ..., ik} in which case we store I1 in the home grid cell (x^{th} data bucket) and I2 in an overflow bucket with address <addr 1>. Assume that grid cell x is the first to be split by an overflow and it chooses dimension 1 to split, the separator table will be:

z-code	separator	dimension	layer	address
x	im	1	0	<addr 1>

Now if I2 changes to be I2 = {im, ..., ih} where h > b, i.e. it needs to be divided into two grid cells: I21 = {im, ..., ij-1} and I22 = {ij, ..., ih}. In order to make a comparison with the separator for the same grid cell z-code, we always split it in the same dimension. If I22 is stored in <addr 2> then the table becomes:

z-code	separator	dimension	layer	address
x	im	1	0	<addr 1>
x	ij	1	1	<addr 2>

where layer = 1 indicates that an overflow has occurred in the first overflow area.

Search a data item ix

Given data item ix the z-code is calculated as z(ix). According to the value of z(ix) the separator table is inspected and the separator is compared with ix. There are three possibilities:

(a) no z-code in the separator table matches z(ix) so ix is in the home grid cell;

(b) when layer = 0, if the j^{th} attribute of ix (j is the dividing dimension) is less than the j^{th} attribute of the separator, then ix is stored in the home grid cell otherwise ix is stored in <addr 1>;

(c) when layer = 1, if the j^{th} attribute of ix (j is the dividing dimension) is less than the jth attribute of the separator, then ix is stored in <addr 1>, otherwise it is stored in <addr 2>.

Appendix A9

Selecting data organisation

(a) Sequential or indexed sequential

When the activity is high the data can be organised by a sequential access method ordered on the dominant attribute identified (usually the primary key). The level at which the system will go to the sequential organisation is a break-even problem. To determine the break-even point the following algorithm is used.

INPUT = { n, m, R, b, k, a }

where

k: the length of the key.

a: address referring to a data item.

R: the average length of the record length.

OUTPUT = { break point: percentage of the activity }

ALGORITHM:

Assume that the activity of a data set is x (%) then the unnecessary access of data in a sequential organisation will be: (1 - x) x n x R (bytes). If we use an index to store the file and the key length in the index is k then (x) x (k + |a|) bytes will be accessed for the index file. Here |a| is the length of the address of the index. Each access to the index results in an access to a data block and therefore, the total extra accesses will be (x) x n x (k + |a|) + (x) x n x (b - 1) x R. The interpretation for the formula is that the first part is the number of bytes to be accessed for the index and the second part is the number of bytes accessed to the required data block minus the wanted one (R). To decide the break-even point for a sequential organisation we have the inequality:

$(1 - x) \times n \times R < (x) \times n \times (k + a) + (x) \times n \times (b - 1) \times R$

That is:

$(1 - x) \times R < (x) \times (k + a) + (x) \times (b - 1) \times R$

$R - (x) \times R < (x) \times k + (x) \times a + (x) \times b \times R - (x) \times R$

$(x) \times (k + a + b \times R) > R$

Solving the equation, we get:

$$Bp1 = (x > \frac{R}{k + a + b \times R} = \frac{R}{c + b \times R}$$

Here C is the index storage overhead.

The Bp1 is a break-even point condition. If Bp1 is true and the access mode is not crucial then sequential organisation can be used. If Bp1 is not true then index organisation is to be used.

This is the function for making a decision on a sequential file if real-time mode is not required by the user.

Thus the rule is:
if (bp1 = true) and (access mode ≠ real time) then sequential.

(b) Indexed sequential or hashing

When access time is important either the hashing or indexed sequential algorithm can be chosen. The decision relies mainly on the following factors: (1) available min. memory M; (2) data vitality Dyn; (3) data distribution Ds. Rules governing the decision can be described as:
if ((|index| ≤ M) and (Ds is even))then indexed sequential;
if ((|index| > M) and (Ds is even) and (Dyn is low)) then hashing;
if ((|index| > M) and (Dyn is uneven) then further factors have to be considered. The factors can be the application requirements such as storage utilisation Su and access speed.

Appendix 10

Examples

Example 1

Rules regarding the algorithm selection have been described in section 4.1. Here, we illustrate how those rules will work by giving some examples. In the following two simple examples, we shall demonstrate the usage of the system by applying elimination rules for the selection of an access algorithm.

Assumption

memory size : M = 1,024 K,

short lived data set : tm ≤ 4 weeks.

(1) Given features of an application

Data set : D = { d1, d2, ..., dn},

Life span : tm ≤ 2 weeks.

Selection

According to elimination rule ER2, ALT[1] (EXCELL algorithm) is selected.

(2) Given features of an application :

Data set : D = { d1, d2, ..., dn}

Life span : tm > 4 weeks

The derived features

index file size : |Index| < 1,024 K

|Index| = |Key| x |G|

where

|Key| is the length of the key

|G| is the number of grid partitions for the data set without causing overflow.

Selection

According to elimination rule ER1, ALT[1] (EXCELL algorithm) is selected.

Explanation

Elimination rules are constructed to select an access algorithm matching data sets with salient features. These features are dominant factors in the selection of a specific access algorithm. The justification of the elimination rules is to avoid intensive rule searches simply because a further examination is not cost-effective (for example, the data set is small or the data set is short-lived), or the suggested algorithm does match those salient features of the considered data set (for example, the z-hashing is suitable for data sets with even data distribution and static features, the BANG file is extremely good for dynamic data sets).

Choosing an algorithm by initial selection rules

Example 2

The following is a more complex example to explain how an initial algorithm selection is made using the system.

(1) Given features of an application:

data set	:	D = { (0, 1), (0, 2), (0, 3), (1, 6), (2, 2), (2, 3), (2, 7), (3, 1), (3, 2), (3, 3), (3, 6), (3, 7), (4, 3), (6, 3), (6, 5) }
bucket size	:	b = 4
index and memory size:		|Index| > M
data distribution	:	Deven = (4 + 0 + 3 + 2) / 16 = 56.25 %
dynamic factor	:	Dyn < 40%
range search rate	:	Rs < 30%

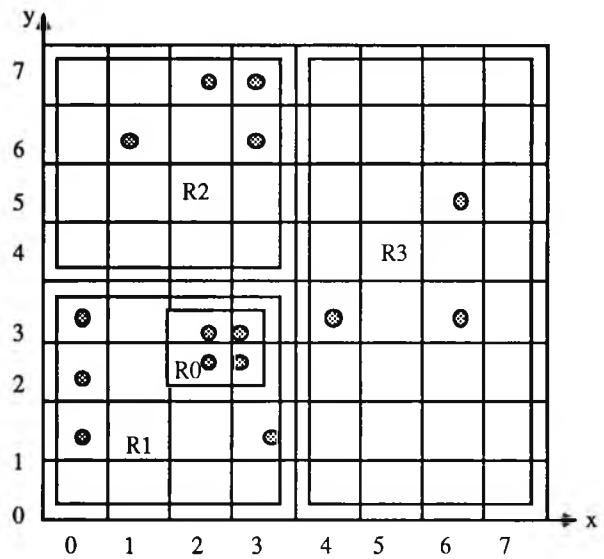


Figure 5. The given data set uses the BANG-file algorithm.

Using initial algorithm selection rules we will get:
h(ATR1) = 0 + 0 + 0 + 1 = 1
h(ATR2) = 1 + 0 + 0 + 0 = 1
h(ATR3) = 1 + 0 + 0 + 0 = 1
h(ATR4) = 1 + 0 + 0 + 0 = 1
h(ATR5) = 1 + 1 + 1 + 0 = 3
h(ATR6) = 1 + 0 + 0 + 0 = 1
thus the BANG-file is chosen.

Performance evaluation for the BANG-file access algorithm

Storage utilisation

minimal requirement = $\lceil n/b \rceil = \lceil 15 / 4 \rceil = 4$

actual requirement = $|R| = 4$

where R = {R0, R1, R2, R3 }

Hence Su = minimal requirement / actual requirement = 100%

Speed

Point search :

assume 50% of the times the required index is in main memory and other 50% in the secondary storage (disk):

Tp = 1.5 x Tsec

Range search accuracy :

range search accuracy depends on the individual selection criteria. In this evaluation we assume a few selections and calculate the average.

$$\text{accuracy } a = \frac{\text{number of data item required}}{\text{actual number of data buckets searched x bucket size b}}$$

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 9 / (4 x 4) = 56.25 %

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 4/ (2 x 4) = 50%

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

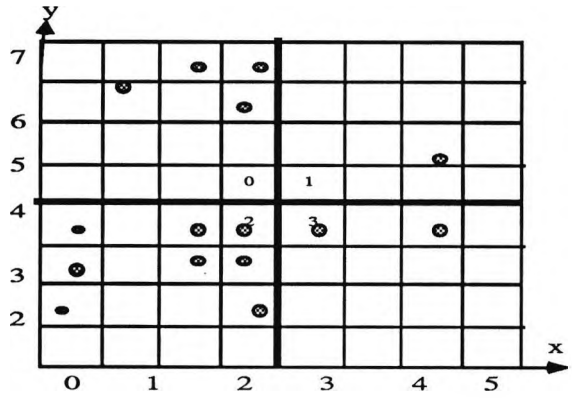
a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 = 47.92%

Comparing with other algorithms

(1) Performance evaluation for EXCELL algorithm

(a) with overflow handling



(b) without overflow handling

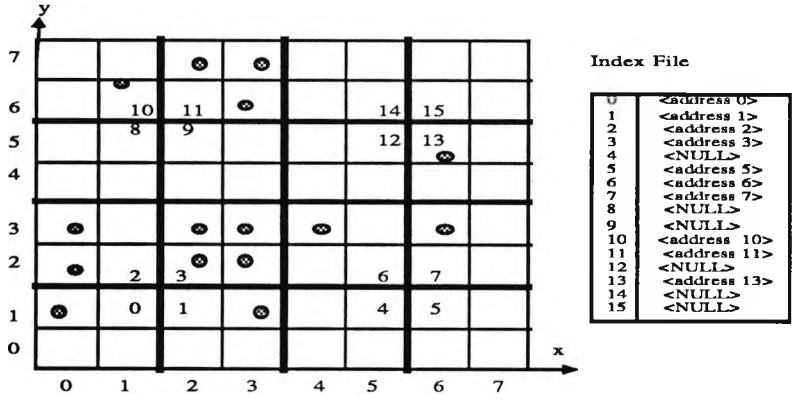


Figure 6. The given data set applies EXCELL algorithm.

Notes:
the EXCELL needs more space to store the index than the BANG-file.

Storage utilisation

(a) with overflow handling

minimal requirement = $\lceil n/b \rceil = \lceil 15 / 4 \rceil = 4$

actual requirement = 5 (1 for overflows, 4 for the partition)

Hence Su = 4/5 = 80%

(b) without overflow handling

actual requirement = 9

Hence Su = 4/9 = 44%

Speed

Point search :

assume 50% of the times the required index is in main memory and other 50% on the disk:

Tp = 1.5 x (4 + 1 + 2 + 4 + 4 x 2) / 15 = 1.9 x Tsec

Range search accuracy :

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 9 / (5 x 4) = 45 %

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 4/ (2 x 4) = 50%

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 44.17 %

(2) Performance evaluation for the z-hashing algorithm

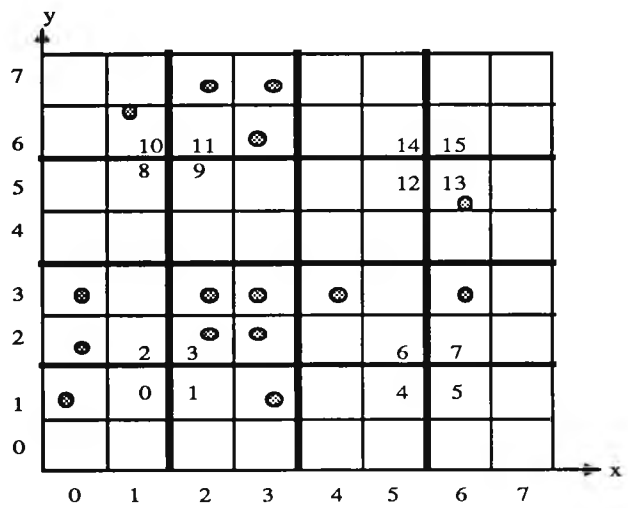


Figure 7. The given data set employs the z-hashing algorithm.

Notes: z-hashing needs no index.

Storage utilisation

minimal requirement = $\lceil n/b \rceil = \lceil 15 / 4 \rceil = 4$

actual requirement = 16

Hence Su = 4/16 = 25%

Speed

Point search :

Tp = 1 x Tsec

Range search accuracy :

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 9 / (9 x 4) = 25%

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 4/ (3 x 4) ≈ 33 %

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 31.83 %

(3) Performance evaluation for the quantile-/PLOP-hashing

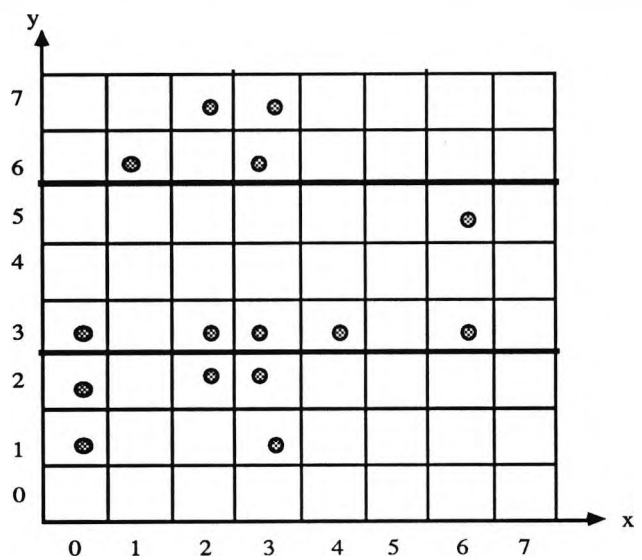


Figure 8. The given data set uses the quantile-/PLOP-hashing algorithm.

Storage utilisation

minimal requirement = $\lceil n/b \rceil = \lceil 15 / 4 \rceil = 4$

actual requirement = 7

Hence Su = 4/7 ≈ 57%

Speed

Point search :

(1) assume indices are stored in main memory the point search is fast, i.e.

Tp = 1 x Tsec

(2) if those indices cannot be stored in main memory, extra disk access is required, on average:

Tp = 1.5 x Tsec

Range search accuracy :

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 9 / (5 x 4) = 45 %

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 4/ (2 x 4) = 50%

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 44.17 %

Notes: when PLOP-hashing destroys the z-order the range search accuracy may deteriorate.

Comparison	**Su**	**Tp**	**a**
BANG-file	**100%**	**1.5**	**47.92%**
Excell	67%	1.9	44.17%
Z-hashing	25%	1	31.83%
Qantile-/PLOP-hashing	66%	1/1.5	44.17%/≤ 44.17%

This example shows that range search is a desirable feature of the data set (as Rs < 30%) and Su is an important factor (since Deven > 50%). Furthermore, the dynamic factor Dyn is relatively high (Dyn < 40%). Consequently, BANG-file offers excellent

performance both in storage utilisation and range search accuracy. In addition, BANG-file copes well with dynamic situations. The performance evaluation has shown a match between the near-optimal performance among available algorithms and the one chosen by the "initial selection rules", i.e. by performance comparison. By comparison the BANG-file is the near-optimal solution. This process heuristically validates these rules used in the algorithm selection process. It can be noticed that z-hashing has excellent point search ability but the cost of dealing with dynamic situations is high and the storage utilisation, low.

Selecting an algorithm by a similarity comparison

Example 3

The following example exercises the similarity comparison element of the rule base.

Assumption

Abstract Application Profile (AAP) has the following features (see Figure 5.)

(1) Data set

D' = { (0 ,1), (0, 2), (0, 3), (1, 6), (2, 2), (2, 3), (2, 7), (3, 1), (3, 2), (3, 3), (3, 6), (3, 7), (4, 3), (6, 3), (6, 5) }

(2) Data distribution by number of overflowed data items in each grid cell (r = 4)

C'(o)[0] = 4
C'(o)[1] = 0
C'(o)[2] = 0
C'(o)[3] = 0

(3) Even data degree

d'(even) = (4 + 0 + 3 + 2) / 16 = 56.25%

(4) Number of overflow grid cells

N'over = 1

(5) Dynamic factor

D'yn < 40%

(6) Point search speed

P's < 70%

(7) Number of empty grid cells

N'empty = 0

(8) Range search rate

R's < 30%

(9) Local data density

L'dx[1] = 12/ 2 = 6 L'dx[2] = 3/2 = 1.5

L'dy[1] = 10/2 = 5 L'dy[2] = 5/2 = 2.5

(10) Query frequency

F'x = 55% F'y = 45%

(11) The access algorithm used is BANG-file.

(12) The stored performance

Storage utilisation

Su = 100%

Speed

Point search :

(1) the index that can be stored in main memory:

Tp = 1 x Tsec

(2) the index that cannot be stored in main memory

Tp = 1.5 x Tsec

Range search accuracy :

average a = (a1 + a2 + a3) / 3 ≈ 47.92%

The given application profile (AP)

(1) Data set

D = { (0 ,2), (0, 3), (1, 1), (1, 3), (1, 4), (2, 2), (2, 3), (2, 7), (3, 2), (3, 3), (3, 7), (4, 1), (4, 3), (4, 4), (6, 4), (6, 5) }.

(2) Data distribution

C(o)[0] = 4
C(o)[1] = 0
C(o)[2] = 0
C(o)[3] = 0

(3) Even data degree

d(even) = (4 + 2 + 1 + 1) / (4 x 4) = 50%

(4) Number of overflow grid cells

Nover = 1

(5) Dynamic factor

Dyn < 40%

(6) Point search speed

Ps < 70%

(7) Number of empty grid cells

Nempty = 0

(8) Range search rate

Rs < 30%

(9) Local data density

Ldx[1] = 11/ 2 = 5.5 Ldx[2] = 5/2 = 2.5

Ldy[1] = 10/2 = 5 Ldy[2] = 6/2 = 3

(10) Query frequency

Fx = 55% Fy = 45%

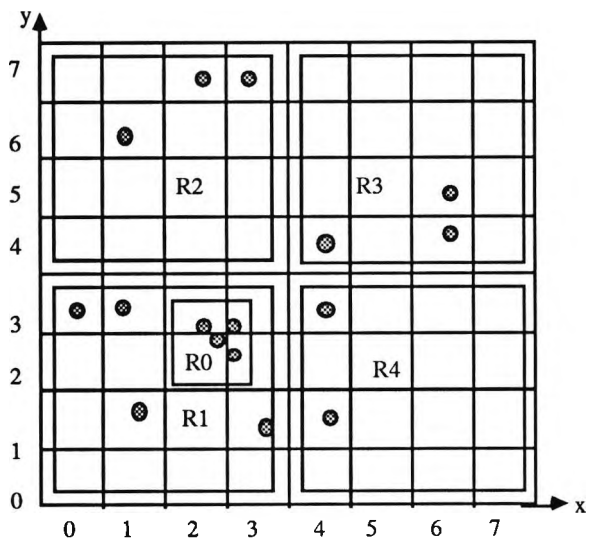


Figure 9. The given data set deploys the BANG-file algorithm.

Similarity computation

Calculated deviations

(1) Data distribution

$$DD = \frac{\sum_{i=0}^{i=3} | C'(o)[i] - C(o)[i])|}{r \times b} = 0$$

(2) Even distribution degree

$$ED = \frac{|d'(even) - d(even)|}{r \times b} = 1.56\ \%$$

(3) Number of overflow grid cells

$OV = |N'over - Nover| / r = 0$

(4) Dynamic factor

$DF = |D'yn - Dyn| = 0$

(5) Point search rate

$DP = |P's - Ps| = 0$

(6) Range search rate

$DR = |R's - Rs| = 0$

(7) Number of empty grid cells

$OE = |N'empty - Nempty| = 0$

(8) Local data density

$$LD = \frac{\sum_{x=1}^{x=s_x} |Ld'(x) - Ld(x)|}{s_x \times b} + \frac{\sum_{y=1}^{y=s_y} |Ld'(y) - Ld(y)|}{s_y \times b} = 25\%$$

(9) Query frequency

$$FD = \sum_{i=x1}^{xm} |f'i - fi| = 0$$

(10) Similarity degree

$s(x) \approx 0.78$

Assuming that $s(x) = 0.78$ is a satisfactory similarity degree, the BANG-file algorithm will be employed for the application profile AP.

Performance evaluation using the BANG-file algorithm for an AP

Storage utilisation

minimal requirement $= \lceil n/b \rceil = \lceil 16 / 4 \rceil = 4$

actual requirement = 5

Hence Su = 4/5 = 80%

Speed

Point search :

(1) index which can be stored in main memory

Tp = 1 x Tsec

(2) index which cannot be stored in main memory

Tp = 1.5 x Tsec

Range search accuracy :

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 11 / (5 x 4) = 55%

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 3/ (2 x 4) = 37.5%

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

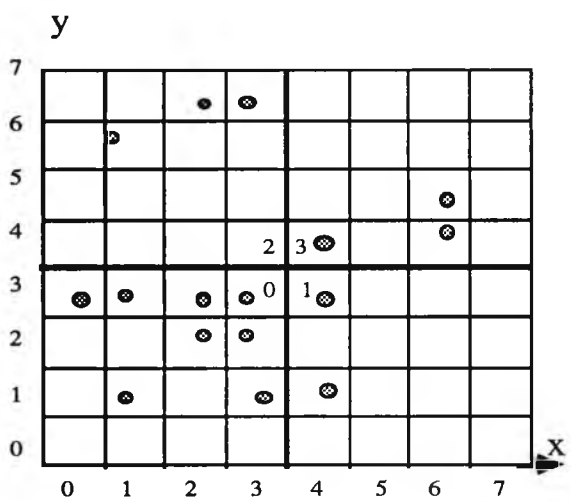
a3 = 3 / (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 43.3%

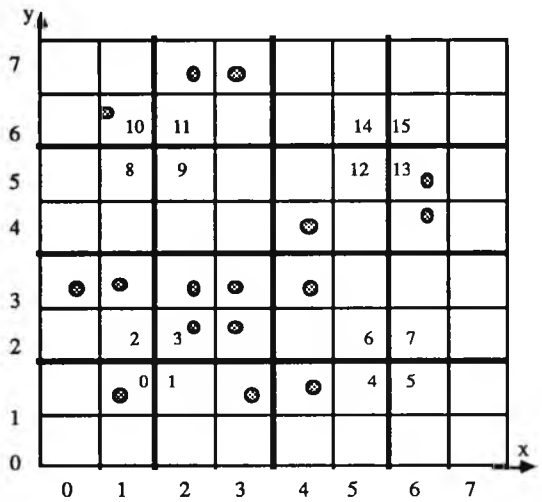
Comparing with other algorithms

(1) Performance evaluation for EXCELL algorithm

(a) with overflow handling



(b) without overflow handling



Index File

0	<address 0>
1	<address 1>
2	<address 2>
3	<address 3>
4	<address 4>
5	<NULL>
6	<address 6>
7	<NULL>
8	<NULL>
9	<NULL>
10	<address 10>
11	<address 11>
12	<address 12>
13	<address 13>
14	<NULL>
15	<NULL>

Figure 10. The given data set applies EXCELL algorithm.

Storage utilisation

(a) with overflow handling

minimal requirement = $\lceil n/b \rceil = \lceil 16 / 4 \rceil = 4$

actual requirement = 5

Hence Su = 4/5 = 80%

(b) without overflow handling

actual requirement = 10

Hence Su = 4/10 = 40%

Speed

Point search :

(1) the index is in main memory

Tp = 1 x Tsec

(2) the index is not in main memory

Tp = 1.5 x Tsec

Range search accuracy :

(a) with overflow handling

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 9 /(5 x 4) = 45%

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 3 / (2 x 4) = 37.5%

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 40 %

(b) without overflow handling

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 9 /(7 x 4) ≈ 39%

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 3 / (2 x 4) = 37.5%

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 38 %

(2) Performance evaluation for the z-hashing algorithm

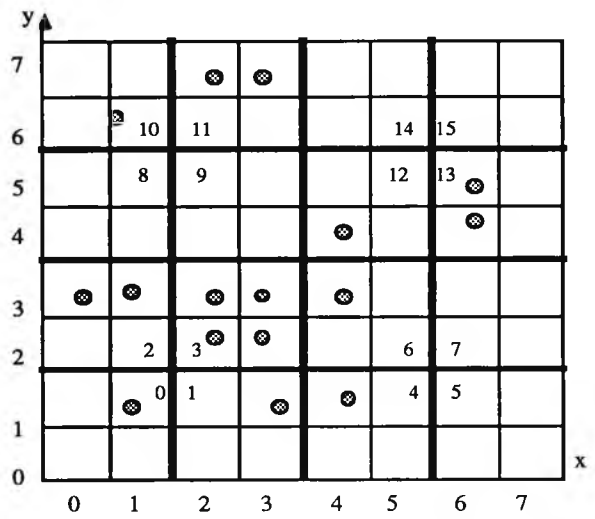


Figure 11. The given data set uses the z-hashing algorithm.

Storage utilisation

minimal requirement = $\lceil n/b \rceil = \lceil 16 / 4 \rceil = 4$

actual requirement = 16

Hence Su = 4/16 = 25%

Speed

Point search :

Tp = 1 x Tsec

Range search accuracy :

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 11 / (9 x 4) ≈ 30.6 %

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 3/ (3 x 4) = 25%

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 31 %

(3) Performance evaluation for the quantile-/PLOP-hashing algorithm

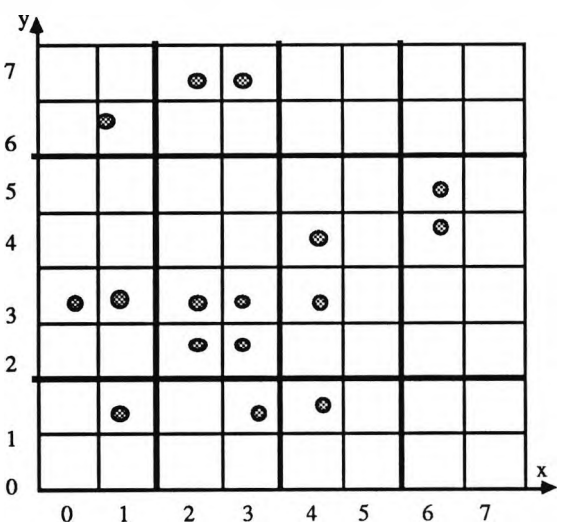


Figure 12. The given data set applies quantile-/PLOP-hashing algorithm.

Storage utilisation

minimal requirement = $\lceil n/b \rceil = \lceil 16 / 4 \rceil = 4$

actual requirement = 9

Hence Su = 4/9 ≈ 44%

Speed

Point search :

(1) assuming indices are stored in main memory, the point search will be fast, i.e.

Tp = 1 x Tsec

(2) if those indices cannot be stored in main memory, extra disk access is required, on average:

Tp = 1.5 x Tsec

Range search accuracy :

(1) select (x, y) where $x \leq 5$ and $y \leq 5$

a1 = 11 / (6 x 4) ≈ 45.8 %

(2) select (x, y) where $x \leq 5$ and $y \geq 5$

a2 = 3/ 2 x 4) = 37.5 %

(3) select (x, y) where $2 \leq x \leq 3$ and $y \leq 3$

a3 = 3/ (2 x 4) = 37.5%

average a = (a1 + a2 + a3) / 3 ≈ 40.25 %

The performance comparison between AAP and AP:

		Su	Tp	a
AAP		100%	1/1.5	47.92%
AP	**BANG-file**	**80%**	**1/1.5**	**43.3%**
	EXCELL	40%/80%	1/1.5	38%/40%
	z-hashing	25%	1	31%
	quantile-/PLOP-hashing	33%	1/1.5	40.25

Here we illustrated a simple example which uses the similarity comparison to choose an access algorithm. The principle is that , if the similarity is identified between a given application profile and the abstract application profile, the algorithm used for the AAP will be used for this application.

Several examples have been given to illustrate how an algorithm is chosen by applying the rule base. Performance is evaluated for various access algorithms. The evaluation results have shown that the selected algorithm is a near-optimal algorithm. Hence the heuristics applied to the rule base are justified.

References

[AG88] R. Agrawal et al.
" Efficient Search in Very Large Databases ", Proc. of the 14th VLDB conference, 1988, Pages 407- 418

[AL87a] M. Allerhand
"Knowledge-Based Speech Pattern Recognition ", 1987, Kogan Page

[AL87b] K. Allgeyer
" Expert Systems Based Configuration of VSAM Files ", Proceedings of Third International Conference on Data Engineering, 1987, Pages 150 - 156

[AN88] K.M. Andress et al.
" Evidence Accumulation & Flow of Control in a Hierarchical Spatial Reasoning System ", AI Magazine, Summer 1988, Pages 75-94

[AN85] A.D. Angelo, et al.
" A Mechanism for Representing and Using Meta-Knowledge in Rule-Based System , Approximate Reasoning in Expert System, edited by M. M. Gupta, et al., North-Holland, 1985, Pages 731-742

[AP85] L. Appelbaum et al.
" ARIES: An Approximate Reasoning Inference Engine ", Approximate Reasoning in Expert System, edited by M.M. Gupta et.al, North-Holland, 1985, Pages 731-742

[AV85] E. Avni et al.
" Software Relational Data base in CAD/CAE and Expert Systems", Approximate Reasoning in Expert System, edited by M.M. Gupta et.al, North-Holland, 1985, Pages 573-592

[BA88] H.S. Baird
"Applications of Multidimensional Search to Structural Feature Identification", NATO ASI Series, Vol. F45, Syntactic and Structural

Pattern Recognition, Springer-Verlag Berlin Heidelberg, 1988, Pages 137 - 143

[BE81] J.C. Bezdok
Pattern Recognition with Fuzzy Objective Function Algorithms, PLENUM Press, 1981, ch3,5,6

[BE89a] C. Beardon
Artificial Intelligence Terminology, 1989

[BE89b] E. Bertine
"Indexing Techniques for Queries on Nested Object", IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No.2, June 1989

[BE88] D. A. Bell et al.
" Clustering Related Tuples in Databases ", The Computer Journal, Vol. 31, No. 3, 1988, Pages 233 - 258

[BE87] A.C. Beerel
" Expert Systems: Strategic Implications and Applications ", ELLIS HORWOOD LIMITED, 1987

[BE81] J.C. Bezdek
" Pattern Recognition with Fuzzy Objective Function Algorithms", Plenum Press, New York and London, 1981

[BE79] J.L. Bently, J.H. Friedman
" Data Structures for Range Searching", Computing Surveys, Vol. 11, No. 4, Dec. 1979, Pages 397 - 409

[BO89] L. B. Booker
"Classifier Systems and Genetic Algorithms", Artificial Intelligence, Vol. 40, No. 1-3, Netherland, September 1989, Pages 235-282

[BO85] L.Bourrelly et al.
" A Formal Approach to Analogical Reasoning ", Approximate Reasoning

in Expert System, edited by M.M. Gupta et.al, North-Holland, Pages 87-104, 1985

[BO83] A. Bolour
" Optimality Properties of Multiple-Key Hashing Functions ", Journal of the Association for Computing Machinery, Vol. 26, No. 2, 1983, Pages 196 - 210

[BR90] Edited by P. B. Brazdil et al.
Machine Learning, Meta-reasoning and Logic, Kluwer Academic Publisher, 1990

[BR73] J. Bronowski
The Accent of Man, Chapter 11, 1973, Pages 353 - 374

[BU84] B.G. Buchanan
Rule-based Expert System: The MYCIN Experiments of the Stanford Heuristic Programming Project, Addison-Wesley Publishing Company, 1984, Pages 209 - 232

[BU89] L. Buisson
"Reasoning on Space with Object-Centered Knowledge Representations", Lecture Notes in Computer Science, Vol. 409, Springer-Verlag, 1989, Pages 325-344

[BU83] W.A. Burkhard
" Interpolation-Based Index Maintenance ", BIT 23, 1983, Pages 274-294

[BU79] W.A. Burkhard
" Partial-Match Hash Coding: Benefit of Redundancy ", ACM trans. on TODS, vol. 4, no. 2, June 1979, Pages 228 - 239

[CA84] V. Cantoni
" Shape Recognition Using Hough Transform ", Cybernetic Systems: Recognition, Learning, Self-Organisation, 1984, Pages 121 - 128

[CA86] M.J. Carey et al.
" Object and File Management in the EXODUS Extendible Database Systems ", Proceedings of the 12th Int. Conf. on VLDBs, 1986, Pages 91 - 100

[CH89] B. Chandrasekaran et al.
" Explaining Control Strategies in Problem Solving ", IEEE Expert Spring 1989, Pages 9 - 24

[CH88a] A. Chandra
" Theory of Database Queries", Proc of 7th ACM Symp. on principles of Database Systems", 1988, Pages 1-9

[CH88b] H.I. Christensen et al.
On Token-matching in Real Time Motion Analysis, Lecture Notes in Computer Science, Vol. 301, Conference: BPRA 4th International Conference on Pattern Recognition, Cambridge (GB), March, 1988, Pages 448-457

[CH86] J.M. Chassery
"Expert Systems, Image Processing and Image Interpretation", Eighth International Conference on Pattern Recognition, 1986, Pages 175-183

[CH84] Y. C. Cheng et al.
"Waveform Correlation Using Tree Matching", 7th Conference Proceeding on Pattern Recognition, 1984, Pages 350 - 354

[CL86] W.J. Clancey
"Heuristic Classification", Knowledge Based Problem Solving, Edited by Jannzy Knoalik, Prentice Hall, 1986, Pages 1 - 67

[CO87a] R. Cole
" Partitioning Point Sets in Arbitrary Dimension ", Theoretical Computer Science, Vol. 49, No. 2, 3, 1987

[CO87b] Douglas W. Cornell et al.
"A Vertical Partitioning Algorithm for Relational Databases", CH2407-5/87/IEEE

[CO78] D. Comer
"The difficulty of Optimum Index Selection", ACM trans. on DB Systems, 1978, Pages 440-447

[GO77] V. Gordon J. et al.
"Two Stage Template Matching", IEEE Trans. Computer, Vol.C-26, 1977, Pages 384-393

[CS88] J. Csirile et al.
"Longest k-distance substrings of two strings", 9th International Conference on Pattern Recognition, Nov. 1988

[DA82] D. Daniel et al.
" An Introduction to Distributed Query Compilation in R*" in Distributed Databases, Edited by H. J. Schneider, North-Holland, 1982, Pages 291 - 310

[DA88a] M. Davison
" The Matching Law ", 1988, Ch2, 4, 5, 12, Lawrence Erlbaum

[DA88b] E.R. Davies
"Tradeoffs between Speed and Accuracy in Two-stage Template Matching", Signal Processing, Vol. 15, Dec. 1988, Pages 351-363

[DA87] R.S. Davis et al.
"Multikey Access Methods Based on Superimposed Coding Techniques", ACM Trans. on Database Systems, Vol. 12, No.4, Dec. 1987, Pages 655-696

[DA79a] R. Davis
"Interactive Transfer of Expertise: Acquisition of New Inference Rules", Artificial Intelligence 12, 1979, Pages 121- 157

[DA79b] L.S. Davis
"Shape Matching Using Relaxation Techniques", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-1, No. 1, January, 1979, Pages 60-72

[DE78] A.K. Dewdney
"Analysis of a Steepest-descent Image-matching Algorithm", Pattern Recognition 10, Pages 31-39, 1978

[DU85] H.C. Du
" On the File Design Problem for Partial Match Retrieval ", IEEE Trans. on Software Eng. Vol. SE-11, No. 2, Feb. 1985, Pages 213 - 222

[EA83] C. M. Eastman
" Current Practice in the Evaluation of Multikey Search Algorithms ", ACM Proc. of 6th annual Int. SIGIR Conf, Vol. 17, No.4, 1983

[EG89] M.J. Egenhofer
"A Topological Data Model for Spatial Databases", Lecture Notes in Computer Science, Vol. 409, Springer-Verlag, Pages 271-286

[EN88] R.J. Enbody
" Dynamic Hashing Schemes" , ACM Computing Surveys, Vol. 20. No. 2. June 1988

[ET88] D.W. Etherington
Reasoning with Incomplete Information, Pitman, London, 1988

[FA86a] O. D. Faugeras et al.
" The Presentation, Recognition, and Partitioning of 3-D Shapes from Range Data ", Techniques for 3-D machine perception, Edited by A. Rosenfeld, 1986, Pages 13 - 51

[FA86b] C. Falousos
" Multiattribute Hashing Using Gray Codes ", 1986 ACM SIGMOD

Record, Pages 227-238

[FA79] R. Fagin
" Extendible Hashing - A Fast Access Method for Dynamic Files ", ACM trans. on TODS, vol. 4, no. 3, Sep. 1979, Pages 315 - 344

[FE88] J. H. Fetzer
" Program Verification ", Communications of ACM, Vol. 31, No. 9, 1988, Pages 1048-1063

[FI73] M.A. Fischler et al.
"The Presentation and Matching of Pictorial Structures", IEEE Trans. C-22, 1973, Pages 67-92

[FO88] C.L. Forgy
"Rete: A Fast Algorithm for the Many Pattern/Many Object", Pattern Match Problem, Readings in Artificial Intelligence and Databases, 1988, Pages 547-557

[FO82] C.L. Forgy
"Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, Pages 17 - 37, 1982

[FR89] M.W. Freeston
"A Well-Behaved File Structure for the Storage of Spatial Objects, Lecture Notes in Computer Science, Vol. 409, Springer-Verlag, 1989

[FR88] M. W. Freeston
" Grid Files for Efficient PROLOG Clause Accesses " , PROLOG and Databases, Editor P. M. D. Gray, 1988, Pages 188 - 211

[FR87] M. W. Freeston
" The BANG File: A New Kind of Grid File ", 1987 Annual Conf., SIGMOD Record, Vol. 16, No.3, Dec. 1987, Pages 260 - 269

[FR86a] M. W. Freeston

"Data Structures for Knowledge Bases : Multi-dimensional File Organisation", Technical Report TR-KB-13, 22 August, 1986

[FR86b] R.A. Frost
"Introduction to Knowledge Base Systems", Collins, 1986, Chapter 10

[GA79] J. Gaschnig et al.
"Preliminary Performance Analysis of the Prospector Consultant System for Mineral Exploration", In proceedings of Sixth International Joint Conference on Artificial Intelligence, 1979, Pages 308-310

[GA89] G. Gardarin, et al.
" Managing Complex Objects in an Extensible Relational DBMS", Proc. of the 15th International Conference on VLDB, 1989, Pages 45-53

[GH89] W. W. Chang
" A Signature Method for the Starbust Database System ", Proc. of the 15th International Conference on VLDB, 1989, Pages 145-153

[GO88] G.H. Gonnet, Per-Ake Larson
" External Hashing with Limited Internal Storage ", Journal of the Association for Computing Machinery, Vol. 35, No.1, Jan. 1988, Pages 161-184

[GO84a] A. Goshtasby et al.
"Image Matching by a Probabilistic Relaxation Labelling Process", 7th International Conference on Pattern Recognition, 1984, Pages 307-309

[GO84b] Y. I Gold
"On the Order of Examining Data-point in SSD Template Matching", 7th International Conference on Pattern Recognition, 1984, Pages 1077-1080

[GO77] Y. I. Gold
" On the Order of Examining Data-points in SSD Template Matching", 7th International Conference on Pattern Recognition, July, 1984, Pages 1077

- 1080

[GR90] P.M. Griffin et al.
"A Methodology for Pattern Matching of Complex Objects", Pattern Recognition, Vol.23, No. 3/4 Pages 245-154, 1990

[GU89] O. Gunter
"The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases", 1989 IEEE Conference on Data Engineering, Pages 598-605

[GU84] A. Guttman
"R-Trees: A Dynamic Index Structure for Spatial Searching", Proceedings of ACM SIGMOD Int. Conf. on Management of Data, 1984, Pages 47 - 56

[HA89] N.I. Hachem
"Key-structural Access Methods for Very Large Files Derived from Linear Hashing", 1989 IEEE Conference on Data Engineering, Pages 305-312

[HA83] F. Hayes-Roth et al.
" Building Expert Systems", 1983

[HE89] A. Henrich
" The LSD Tree: Spatial Access to Multidimensional Point and Non-point Objects ", Proc. of the 15th International Conference on VLDB, Pages 45-53

[HE88] M. Henri et al.
"Elastic Matching Versus Rigid Matching by Use of Dynamic Programming", 1988

[HE80] C.F. Herot
" Spatial Management of Data", ACM trans. on TODS, vol. 5, no. 4, Dec. 1980, Pages 493 - 514

[HO90] J.P.E. Hodgson
"Automatic Generation of Heuristics", Formal Techniques in Artificial Intelligence, Edited by R.B. Banerji, ELASEVIER SCIENCE PUBLISHERS, 1990, Pages 123-172

[HO89a] A. Hopgood
"An Inference Mechanism for Selection and Its Application to Polymers", Artificial Intelligence in Engineering, 1989, Pages 197

[HO89b] E. Hollnagel
The reliability of Expert Systems, Ellis Horwood Ltd., 1989, Pages 168-223, Pages 173-183

[HO88] J. Manray Holt
"A Fast Binary Template Matching Algorithm for Document Image Data Compression", Lecture Notes in Computer Science, Vol. 301, Conference: BPRA 4th International Conference on Pattern Recognition, Cambridge (GB), 28-30 March 1988

[HU88a] A. Hutflesz et al.
"Globally Order Preserving Multidimensional Linear Hashing ", 14th International Conference on Data Engineering, 1988, Pages 572 - 579

[HU88b] Andreas Hutflesz et al.
" Twin Grid Files: Space Optimisation Access Schemes ", ACM SIGMOD. International Conference on Management of Data, Vol. 17. No.3, Sep. 1988, Pages 183 - 190

[IE85] IEEE CS Press
Entity-Relationship Approach: The Use of ER Concept in Knowledge Presentation, IEEE CS Press/North Holland, 1985, Pages 140 - 169

[JA89] P. Jakson
Logic-Based Knowledge Representation, MIT Press Series in Logic Programming, 1989

[JO84] Journal of Algorithms
Heuristic Matching for Graph Satisfying the Triangle Inequality, 1984

[JO89] R. Johns
"Leave it in Expert Hands", Computing, Vol. , No. 26 Oct. 1989

[KA85] K. Kawagoe
" Modified Dynamic Hashing", 1985 ACM SIGMOD Record, Pages 201 - 213

[KE89] L. Kerschberg
"The Role of Loose Coupling in Expert Database System Architectures", 1989 IEEE Conference on Data Engineering, Pages 255-256

[KH84] N.A. Khan
" Matching an Imprecise Object Description with Models in a Knowledge Base ", Seventh International Conference Proc. on Pattern Recognition, 1984, Pages 1131-1134

[KI89] H.P. Kriegal et al.
"Performance Comparison of Point and Spatial Access Methods", Lecture Notes in Computer Science, 409, SSD'89, Design and Implementation of Large Spatial Database, 1989, Pages 89-114

[KI90] W. Kim
"Object-Oriented Concepts, Databases and Applications", Edited by W. Kim et al., ACM Press, Addison-Wesley Publishing Company, Pages 341-369

[KJ84] P. Kjelltery et al.
" Cascade Hashing ", Proceedings of the 10th International Conference on VLDBs, 1984, Pages 481 - 492

[KO86] H.F. Korth et al.
Database System Concepts, McGRAW-HILL, International Editions, Computer Series, 1986, chapter 6, 8, 9, 11

[KR89] H.P. Kriegal et al.
"Performance Comparison of Point and Spatial Access Methods", Pages 89-114, Lecture Notes in Computer Science, Vol. 409, Springer-Verlag

[KS88a] H.P. Kriegel, B. Seeger
" PLOP-Hashing: A Grid File without Directory ", 14th International Conference on Data Engineering, 1988, Pages 369 -376

[KS88b] H.P. Kriegel, B. Seeger
" Multidimensional Quantile Hashing is Very Efficient for Nonuniform Distribution ", Will Appear in Information Science, 1988

[KS87] H.P. Kriegel, B. Seeger
" Multidimensional Quantile Hashing is Very Efficient for Nonuniform Distribution ", Proceedings of Third International Conference on Data Engineering, 1987, Pages 10 - 17

[KS86] H.P Kriegel, B. Seeger
"Multidimensional Order Preserving Linear Hashing with Partial Expansion ", Proceedings of International Conference on Database Theory, 1986

[LA88] P. A. Larson, et.al
"Linear Hashing with Separators -- a Dynamic Hashing Scheme Achieving One-access Retrieval", ACM trans. on TODS, vol. 13, No. 3, Sep. 1988, Pages 366 - 338

[LA86] C.P. Langlotz
" Using Decision Theory to Justify Heuristics ", Proceedings of fifth national conference on A.I., 1986, Pages 215 - 219

[LA85a] A. Lagomasino et al.
" Imprecise Knowledge Representation in Inferential Activities ", Approximate Reasoning in Expert System, edited by M.M. Gupta et.al, North-Holland, Pages 473-498, 1985

[LA85b] C.E. Langenhop et al.
"An Efficient Model for Representing and Analysing B-trees", 1985 ACM Annual Conference Proceedings, Pages 35 - 40

[LA85c] P. A. Larson, et.al
" External Perfect Hashing ", Proceedings of ACM SIGMOD on Management of Data, 1985, Pages 190 - 200

[LA82] P. A. Larson, P.A.
"A Single File Version of Linear Hashing with Partial Expansion", Proceedings on VLDB, 1982, Pages 300 - 309

[LA80] P. A. Larson,
" Linear Hashing with Partial Expansions ", Proceedings on VLDB, 1980, Pages 224 - 235

[LA78] P. A. Larson, et.al
" Dynamic Hashing ", Bit 18, 2 (1978), Pages 184 - 201

[LE88a] D. L. Lee et al.
" An O(n + k) Algorithm for Ordered Retrieval from an Associative Memory ", IEEE Transactions on Computers, March 1988, Pages 368 - 372

[LE88b] Chin-Hwa Lee et al.
"Partial Matching of Two Dimensional Shapes", 9th International Conference on Pattern Recognition, Nov. 1988

[LE87] J. K. Lee et al.
"Intelligent Stock Portfolio Management System", Expert Systems: The International Journal of Knowledge Engineering, Pages 74-89, 1987

[LE80] D. T. Lee, C. K. Wong
" Quintary Trees: A File Structure for Multidimensional Database Systems ", ACM trans. on TODS, vol. 5, no. 3, Sep. 1980, Pages 339 - 353

[LI89] W. Litwin
" Multilevel Trie Hashing ", Technical Report, 23 pages, Le Chesnay, France, 1989

[LI87] Lindsay et al.
"A Data Management Extension Architecture", Proceedings of the ACM-SIGMOD International Conference on Management of Data, 1987, Pages 220-226

[LI86] X. B. Li et al.
" Image Matching with Multiple Templates ", Proceedings of Computer Society Conference on Computer Vision and Pattern Recognition, 1986

[LO89] D.B. Lomet et al.
" A Robust Multi-attribute Search Structure ", Pages 296-304, 1989 IEEE Conference on Data Engineering

[LO87] D. B. Lomet
" The hB-tree: A Robust Multi-attribute Indexing Method", July 1987, Technical Report TR-87-05, Wang Institute of Graduate Studies

[MA88] M.V. Mannino, et al.
" Statistical Profile Estimation in Database System ", ACM Computing Surveys, Vol. 20, No.3, September 1988

[MA86] L. F. Mackect et al.
" R* Optimiser Validation and Performance Evaluation for Distributed Queries", in Proceedings of 12th International Conference on VLDB, 1986, Pages 149-159

[MI90] D. P. Miranker
TREAT: A New and Efficient Match Algorithm or AI Production Systems, Pitman, London, Morgan Kanfman Publishers, Inc. San Mateo, California 1990

[MI85] L.L. Miller
" Performance of Hash Files in A Microcomputer Based Parallel File System ", 1985 ACM Annual Conference Proceedings, Pages 29 - 34

[MO86] C. Mohan et al.
"Transaction Management in the R*, Distributed Database Management Systems", ACM Transactions on Database Systems, Vol. 11, No. 4, 1986, Pages 378-396

[MO84] B.B. Moshe et al.
"Contextual Template Matching : A Distance Measure for Pattern with Hierarchically Dependent Features", IEEE Trans. Pattern Analysis Machine Intelligence, Vol.PAMI-6, Pages 201-211, 1984

[MU84] J.K. Mullin
" Unified Dynamic Hashing ", Proceedings of the 10th International Conference on VLDBs, 1984, Pages 473 - 480

[NE89] E.J. Neubold
" Coupling Knowledge Based Systems with Large Data Stores ", Pages 257-258, 1989 IEEE Conference on Data Engineering

[NE88] D. Nebendahl, et al.
"Expert Systems", Project Experience with SIUX, Pages 185-209, John Wiley & Sons Limited, 1988

[NG86] G. T. Nguyen
" Object Prototypes and Database for Expert Database Systems ", Proceedings of the First International Conference on Expert Database Systems, 1986, Pages 3-14

[NG85] H.T. Nguyen et al.
" On Foundations of Approximate Reasoning ", Approximate Reasoning in Expert System, edited by M.M. Gupta et.al, North-Holland, Pages 33-45, 1985

[NI84] J. Nievergelt et al.
"The Grid File: An Adaptable Symmetric Multi-key File Structure", ACM trans. on TODS Vol. 9, No. 1, 1984

[OH88] K. Ohmori, et al.
"A Unified View on Tree Metrics", NATO ASI Series, Vol. F45, Syntactic and Structural Pattern Recognition, Springer-Verlag Berlin Heidelberg 1988, Pages 85 - 100

[OM88] E. Omiecinski
" Concurrent Storage Structure Conversion: From B+-tree to Linear Hash File ", 14th International Conference on Data Engineering, 1988, Pages 589 -595

[OO89] B.C. Ooi et al.
"Extending a DBMS for Geographic Applications", Pages 590-597, 1989 IEEE Conference on Data Engineering

[OR89] J.A. Oriensten et al.
"Strategies for Optimising the Use of Redundancy in Spatial Database", Pages 115-136, Lecture Notes in Computer Science, Vol. 409, Springer-Verlag

[OR88a] R. Orlandic
" Compact 0-Complete Trees ", Proceedings of the 14th VLDB Conference, Aug. 1988, Pages 372-381

[OR88b] J.A. Orenstein
" PROBE spatial data modelling and query processing in an image database application ", Technical report, AITD reference No. 155, 38 pages

[OR86] J.A. Orenstein
"Spatial Query Processing in an Object-Oriented Database System", ACM SIGMOD 1986

[OR84] J.A. Orenstein
" A Class of Data Structures for Associative Searching ", Proceeding of 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984, Pages 181-190

[OS89] P.V. Oosterom, et al.
"An Object-Oreinted Approach to the Design of Geographic Information Systems", Pages 255-270, Lecture Notes in Computer Science, Vol. 409, Springer-Verlag

[OT85] E.J. Otoo
" A Multidimensional Digital Hashing Scheme for Files with Composite Keys ", 1985 ACM SIGMOD Record, Pages 214 - 229

[OT84] E.J. Otoo
" A Mapping Function for the Directory of a Multidimensional Extendible Hashing ", Proceedings of the 10th International Conference on VLDBs, 1984, Pages 493- 506

[OT88] E.J. Otoo
" Linearising the Directory Growth in Order Preserving Extendible Hashing ", 14th International Conference on Data Engineering, 1988, Pages 580 - 587

[OW88] O. Owolab et al.
Approximate String Matching: Investigating with a Hardware String Comparator", pages 536-545, Lecture Notes in Computer Science, Vol. 301, Conference: BPRA 4th International Conference on Pattern Recognition, Cambridge (GB), March, 1988

[OZ89] M.T. Ozsu
"From Data Management to Knowledge Management - Prospects for the Next Decades", ACI Congress 89, "Prospective for the 90's", Pages 118-124

[OZ85] E. A. Ozkarahan

"Dynamic and Order Preserving Data Partitioning for DB Machine", Proceedings of the 12th Int. Conf. on VLDBs, 1985 Pages 358 - 368

[PA89] D.J. Parke
"Integrating AI and DBMS Through Stream Processing, Pages 259-261, 1989 IEEE Conference on Data Engineering

[PA86] S. K. Pal
Fuzzy Mathematical Approach to Pattern Recognition, 1986, Wiley Eastern Ltd.

[PA90] D. W. Patterson
Introduction to Artificial Intelligence and Expert Systems, Chapter 10 ("Matching Techniques"), Prentice-Hall, 1990, Pages 188-210

[PA74] T. Pavlidis
" Structural Pattern Recognition ", Ch3, 4, 5, 6, 1974

[PE89] W. A. Perkins
Knowledge Base Verification, Pages 353 - 376, Topics in Expert System Design: Methodologies and Tools, Edited by G. Guida et al., North-Holland, 1989

[PE84] J. Pearl
Heuristics: Intelligent Search Strategies for Computer Problem Solving, 1984, Addison-Wesley

[PE83] M.G. Peter et al.
"Optimisation Algorithm for Distributed Queries", IEEE Transaction on Software Engineering, Vol. SE-9, No. 1, January 1983

[PL84] D. A. Plaisted
"Heuristic Matching for Graphs, Satisfying the Triangle Inequality", Journal of Algorithms, Vol. 5, Pages 163-179, 1884

[PO84] P. Politakis et al.

"Using Empirical Analysis to Refine Expert System Knowledge Bases", Artificial Intelligence in Press, 1984

[PR86] M. Prietula
" Flexible Interfaces and the Support of Physical Database Design Reasoning ", Proceedings of the first International Conference on Expert Database Systems, April 1986, Pages 329 - 342

[RA88] M.V. Ramakrishna
" An Exact Probability Model for Finite Hash Table ", 14th International Conference on Data Engineering, 1988, Pages 362 - 368

[RE89] M. Reeve et al.
Parallel Processing and Artificial Intelligence, 1989, Ch4, Pages 37-49, T.L. Kunii, "Information-driven Pattern Recognition through Communicating Processes - a case study on classification of wallpaper groups"

[RE88] M. Regnier
" Trie Hashing Analysis ", 14th International Conference on Data Engineering, 1988, Pages 377 - 381

[RE84] M. Regnier
" Analysis of Grid File Algorithms ", BIT., 1984

[RO81] J. T. Robinson
" The k-d-B tree: A Search Structure for Large Multidimensional Dynamic Indexes ", Proceedings of ACM SIGMOD Int. Conf. on Management of Data, 1981, Pages 10 - 18

[RU87] W. D. Ruchte
" Linear Hashing with Priority Splitting ", Proceedings of Third International Conference on Data Engineering, 1987, Pages 2 - 9

[SA89] B. Samadi
TUNEX: A Knowledge-Based System for Performance Tuning of the

UNIX Operating System, IEEE Transaction on Software Engineering, Vol. 15, No. 17, July 1989, Pages 861-874

[SA88] H. Samet
"Hierarchical representations of collection of small rectangles", ACM Computer Surveys, Vol. 20, No. 4, December 1988, Pages 271-309

[SA85] G. Salton
" Advanced Information Retrieval Methods ", Proc. of The First PAN Pacific Computer Conference, 1985, Pages 119 - 133

[SC81] M. Scholl
" New File Organisations Based on Dynamic Hashing ", ACM Trans. Database Syst. Vol. 6, No. 1 (Mar. 1981), Pages 194 - 211

[SE87] T. Sellis, et al.
"The R^+-tree: A Dynamic Index for Multi-dimensional Object ",
Proceedings of the 13^{th} VLDB Conference, Briton 1987

[SH91] D. Shasha et al.
"Optimising Equijion Queries In Distributed Databases Where Relations Are Hash Partitioned", ACM Transaction on Database Systems, Vol. 16, No. 2, June 1991, Pages 279-308

[SH88] S. Shekfar et al.
" A Formal Model of Trade-off Between Optimisation and Execution Cost in Semantic Query Optimisation ", Proc. of the 14th VLDB Conf. 88, Pages 457 - 467

[SH87a] L.G. Shapiro et al.
"Ordered Structural Shape Matching with Primitive Extraction by Mathematical Morphology", Pattern Recognition, January, 1987

[SH87b] L.G. Shapiro
"Ordered Structural Shape Matching", Syntactic and Structural Pattern

Recognition, Edited by G. Ferrete st. al, Springer-Verlag Berlin, Heidelberg, 1988

[SH84] M. Sholom et al.
A Practical Guide to Designing Expert Systems, Chapman and Hall Ltd., 1984, Ch 6, 7, Pages 138-169

[SH81] L.G. Shapiro et al.
"Structural Description and Inexact Matching", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-3, No. 5, September 1989, Pages 504-519

[SH78] L.G. Shapiro
"Inexact Matching of Line Drawing in a Syntactic Pattern Recognition System, Pattern Recognition 10, Pages 313-321, 1978

[SI88] H. W. Six et al.
" Spatial Searching in Geometric Databases ", Proceedings of Fourth International Conference on Data Engineering, 1988, Pages 496 - 503

[SN87] J. C. Snader
" Look It Up Faster with Hashing ", January 1987, BYTE, Pages 129 - 144

[SP89] SPIE
International Constant-time Pattern Matching for Real-time Production Systems, Proc. SPIE - International Society of Optical Engineering. Vol. 1095, part2, Pages 971-982, 1989

[SR89] J. Srivastava et al.
"TBSAM: An Access Method for Efficient Processing of Statistical Queries", IEEE Transactions on Knowledge and Data Engineering, Dec. 1989, Vol. 1, No.4

[ST89] M. Stonebraker
"Future Trends in Database Systems", IEEE Transactions on Knowledge

and Data Engineering, March 1989, Vol.1, No. 1, Pages 33 - 44

[ST88] L.Sterling
"A Meta-level Architecture for Expert Systems", Meta-level Architectures and Reflection", edited by P.Maes, D. Nardi, North-Holland

[ST87] A. C. Strangard
Robotics and Artificial Intelligence: An Introduction to Applied Machine Intelligence, Prentice Hall, 1987, Pages 80-112

[ST86] M. Stonebraker et al.
" An Analysis of Rule Indexing Implementations in Database Systems ", Proceedings of the first International Conference on Expert Database Systems, 1986, Pages 353-364

[SY88] S. Abu-Hakima
"Rationale: Reasoning by Explaining", Proceedings of 4th Conf. on Data Engineering, 1988, Pages 258-265

[SZ84] J. L. Szwarcfiter
" Optimal Multiway Search Trees for Variable Size Keys ", ACTA Informatica 21, 1984, Pages 47 - 60

[TA86] D. Tasker
" The Evolution of Data ", CIPS Congress'86, Pages 297 - 302

[TA85] M. Tamminen
" On Search by Address Computation ", BIT 25, 1985, Pages 135-147

[TA82] M. Tamminen
" The Extendible Cell Method for Closest Point Problems", BIT 22, 1982, Pages 27 - 41

[UI89] J.D. Ullman
"Principles of Database and Knowledge-base Systems", Vol. II: The New Technology, Computer Science Press, 1989

[VA89] Edited by S. Vadera
Expert System Applications, Sigma Press, 1989, ch1

[VA86] P. Valduriez, et al.
"Implementation Techniques of Complex Objects", Proceedings of the 12th Int. Conf. on VLDBs, 1986 Pages 101 - 110

[VA84] P. Valduriez, et al.
" A Multi-Hashing Scheme Using Predicate Trees ", Proc. of ACM SIGMOD'84, Vol. 14, No. 2, Pages 107 - 114

[VE89] D. Vet, et al.
"A Practical Algorithm for Evaluating Database Enquiries", Software Practice and Experience, Vol. 19, No. 5, May 1989, Pages 491 - 495

[WA78] D.A. Waterman et al.
Pattern-directed Inference Systems, 1978

[WE89] J.R. Weitzel
"Developing Knowledge-based Systems: Reorganising the System Development Life Cycle", Communication of the ACM, April 1989, Vol. 2, No. 4, p482- 488

[WE83] S.M. Weiss et al.
A Practical Guide to Designing Expert Systems, Pages 138-155, 1983, Chapman and Hall Ltd.

[WH85] T. Whalen et al.
" Goal-directed Approximate Reasoning in a Fuzzy Production System ", Approximate Reasoning in Expert System, edited by M.M. Gupta et.al, North-Holland, Pages 505-518, 1985

[WI89] B.P. Wise et al.
"Evaluation of Uncertainty Inference Models III: The Role of Tuning", Machine Intelligence and Pattern Recognition: Uncertainty in Artificial

Intelligence, Edited by L. N. Karal et al., North-Holland, 1989

[WI87] P. Willet
" Effectiveness of Retrieval in Clustered Document Files ", Proceedings of 11th Int. Online Information Meeting, Dec., 1987

[WI85] D. E. Willard
"New Data Structures for Orthogonal Range Queries ", SIAM J. Comput., Vol. 14, No. 1, Feb. 1985

[WI84] D. N. Willard
" New Trie Data Structures Which Support Very Fast Search Operations ", Journal of Computer and System Science, Vol. 28, Pages 379-394, 1984

[WI84] P. H. Winston
" Artificial Intelligence ", Ch2, Ch11, Ch12, 1984, Addison-Wesley

[WO78] R.Y. Wong et al.
"Sequential Hierarchical Scene Matching, IEEE Trans. C-27, Pages 359-366, 1978

[WU89] X. Wu et al.
" A Knowledge-based Database Assistant", Pages 402-409, 1989 IEEE Conference on Data Engineering

[XU90] L. Xu, et al.
"Improved Simulated Annealing, Boltzmann Machine, and Attributed Graph Matching", Lecture Notes in Computer Science, Vol. 412, 1990, Neural Networks, Pages 151-161, Solving the attributed graph matching problem

[YA79] S. Yamamoto, et al.
" Design of a Balanced Multi-Valued File-Organisation Scheme with the Least Redundancy ", ACM trans. on TODS, vol. 4, no. 4, Dec. 1979, Pages 518 - 530

[YO89] F. F. Yao et al.
"Partitioning Space for Range Queries", SIAM Journal of Computers, Vol. 18, No. 2, Pages 371-384, 1989

[YO89] H. Yokota et al.
"Term Indexing for Retrieval by Unification", Pages 313-320, 1989 IEEE Conference on Data Engineering

[YO74] T.Y. Young
Classification, Estimation and Pattern Recognition, American ELSEVIER Publishing Company, 1974, ch3

[ZA76] L.A. Zadeh
A Fuzzy Algorithmetic Approach to the Definition of Complex or Imprecise Concepts", International Journal of Man-Machine Studies

[ZA71] L.A. Zadeh
A Fuzzy Algorithmetic Approach to the Definition of Complex or Imprecise Concepts", International Journal of Man-Marchine Studies, Vol. 8, 1971

[ZH88] Q. Zhu
"Pattern Classification in Dynamic Environment", Pages 517-526, Lecture Notes in Computer Science, Vol. 301, Conference: BPRA 4th International Conference on Pattern Recognition, Cambridge (GB), March, 1988