DYNAMIC SPIN SCHEMES

By

ZILI FAN

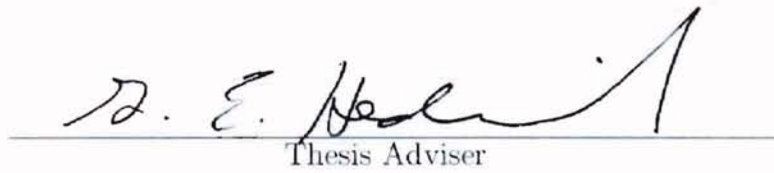
Bachelor of Science
Hangzhou University
Hangzhou, Zhejiang, P.R. of China
1991

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1994

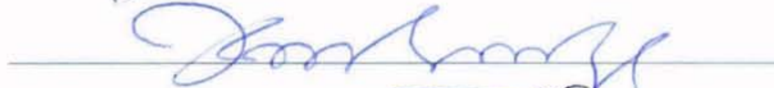
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1996

DYNAMIC SPIN SCHEMES

Thesis Approved:



Thesis Adviser



Dean of the Graduate College

OKLAHOMA STATE UNIVERSITY

PREFACE

This paper presents two dynamic SPIN schemes: Index Dynamic SPIN and Directory Dynamic SPIN. SPIN is a new indexing technique in database design, which is of practical importance in various fields.

The dynamic SPINs modify the static SPINs in two ways: 1) The size of index file is flexible through operations on data. 2) Data overflow and data sparseness are eliminated at the same time. Index Dynamic SPIN introduces the dynamic index tree structures, the shapes of which are dynamic through nodes split and combine. Directory Dyanmic SPIN evolves from extendible hashing, combining with static SPINs properties. It introduces directory structure, instead of index tree in index file. Also, the data file is composed of bucket units, so the flexibility of data file is accomplished in Directory Dynamic SPIN. The performance of the two dynamic SPINs are analyzed, then compared with static SPINs.

I would like to express my sincere gratitude to my major adviser, Dr. G. E. Hedrick, who introduced me to this interesting project, and constantly gave me intelligent guidance. Many thanks also to the other committee members, Dr. K. M. George and Dr. John P. Chandler, for their helpful advisement and suggestions.

I wish to express my sincere gratitude to those who provided suggestions and assistance for this study: Dr. Allen Divall, Mr. Yunpeng Zhang, and Mrs. Ying Fan. My deepest appreciation is extended to my parents whose encouragement and understanding were invaluable throughout the study.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	DEFINITIONS AND TERMINOLOGY	3
III.	STATIC SPIN SCHEMES	5
	The C_SPIN function	5
	The S_SPIN function	7
	The R_SPIN function	10
	The limitations of R_SPIN	11
IV.	DYNAMIC SPIN SCHEMES	15
	Index Dynamic SPIN	15
	Preliminaries	15
	General Description	19
	Space Utilization	23
	Directory Dynamic SPIN	25
	Preliminaries	25
	General Description	26
	Splitting Control	31
	Bucket Structure	33
	Comparisons Between Dynamic SPINs and R_SPIN	34
V.	IMPLEMENTATION	36
	Data Structures	36
	Index Dynamic SPIN	37
	Directory Dynamic SPIN	38
VI.	PERFORMANCE ANALYSIS FOR SPIN	40
	Testing Program	40
	Performance Analysis for R_SPIN	41
	Performance Analysis for D_SPIN	42
	Performance Analysis for T_SPIN	45

Chapter | Page

VII. SUMMARY, CONCLUSION, AND SUGGESTED FUTURE WORK 48

A SELECTED BIBLIOGRAPHY 50

APPENDIX A – GLOSSARY 53

LIST OF FIGURES

Figure		Page
1.	Transformation between different dimensional arrays	7
2.	Transformation values for the array arr[2][4]	9
3.	Tree structures for R_SPIN	11
4.	Different shapes of trees in an index file	16
5.	A leveled structure for the sparse situation	17
6.	Sequential numbers of index for the nodes at the same level	17
7.	A nonsparse (dense) tree	18
8.	Different initial trees	19
9.	Trees with splitted nodes	20
10.	Insertion situation for index dynamic SPIN	21
11.	The final result of the index tree in D_SPIN	23
12.	The whole picture for data indexing and retriev	24
13.	Key transformations in extensible hashing	27
14.	Key transformations in T_SPIN	28
15.	Directory of order d=3 with four buckets	30
16.	T_SPIN transformations	31
17.	Distribution of keys after splitting bucket D	32
18.	Data structure for D_SPIN	37
19.	Experimental results using R_SPIN function	41
20.	The result of experiment using R_SPIN function	42

Figure | Page

21. The result of experiment using D_SPIN function 43

22. An experiment using D_SPIN function 44

23. A search experiment using D_SPIN function 45

24. The result of experiment using T_SPIN function 46

25. The result of experiment using distribution of T_SPIN 47

CHAPTER I

INTRODUCTION

Data indexing is important in database design and management. In large databases the indexing techniques are critical for the fundamental operations of search, insert, update, and delete. Research in this field has generated several structures and algorithms including those for R_trees[1] and for B_trees[2].

Coburn proposed a new data indexing technique, called the Single Point Index Network(SPIN)[4]. The significant difference between SPIN and other data indexing techniques is that SPIN supports layered data relationships using a multidimensional approach. The layered data structure is defined as abstract layers of data using the data structure operations responsible for performing the fundamental operations of search, insert, delete, and update. In an environment of relational databases, such a layered data structure means a data structure supporting internal mappings between fields in different databases. The process of such mappings is also called "horizontal integration [3]," which means the incorporation at one time of information from more than one functional area. Horizontal integration requires that the computer work in multidimensional data spaces and use multidimensional data structures.

According to Coburn[4], SPIN can perform fundamental operations on multidimensional data spaces while other many data structures apparently cannot do the same work with the same speed and flexibility[3]. The SPIN technique evolves from performing a modification to the multidimensional array data structure, because the multidimensional array is the most common multidimensional data structure. SPIN consists of a series of transformation functions which convert multidimensional data

spaces into equivalent linear data spaces. This allows the computer to operate on a multidimensional data space as though it were a linear space. However, the SPIN schemes proposed by Coburn have properties of static structures. They require that data storage space or index storage space be allocated statically. This means that when a file exceeds the allocated space, an overflow happens; consequently, the entire file must be restructured at great expense. In addition, overflow handling schemes often increase the retrieval time as files approach their space limits. To overcome these problems, dynamic SPIN schemes have been proposed. The file structures of dynamic SPINs and their associated algorithms adapt themselves to changes in the size of the file, so expensive periodic database reorganization is avoided. In this thesis, we introduce two dynamic SPIN schemes, index dynamic SPIN and directory dynamic SPIN, with special emphases on the various design issues.

The goals of this thesis are to provide two basic designs of the dynamic SPIN structures and algorithms, to outline some of the techniques that are being developed, to implement the proposed dynamic SPINs, and to show how the various design parameters relate to their performances.

In the thesis, Chapter II defines some technical terms which will be used in the thesis. Chapter III explains static SPIN schemes, including properties of static SPINs and the limitations of static SPINs. Chapter IV describes two dynamic SPIN schemes: index dynamic SPIN and directory dynamic SPIN, some design issues are also discussed in this chapter. Chapter V implements two dynamic SPIN schemes. Chapter VI analyses the performance of dynamic SPINs. Chapter VII is the summary, conclusions and directions for future work. Appendix A is the glossary.

CHAPTER II

DEFINITIONS AND TERMINOLOGY

To make the presentation of dynamic SPINs precise, this chapter defines some terms. Implicit in these definitions is the assumption that the volume of data is large and operations on data are in main storage.

1. *A dense tree* means a tree whose nodes all contain data. In other words, a tree is *full*. Conversely, a *sparse tree* means a tree whose nodes do not all contain data. A majority of its nodes are empty. In database design, *a sparse tree* implies that some storage space is wasted, and *a dense tree* implies that an overflow may happen on the index tree.

2. *static SPIN* means the SPIN package developped by Coburn[3]. This SPIN package includes three basic SPIN schemes: *C_SPIN*, *S_SPIN* and *R_SPIN*. The first two SPINs transform multidimensional data structures into a one dimensional data structures. *R_SPIN*, based on the *S_SPIN*'s transformation, further transforms sparse data structures into dense data structures. All these transformations have static properties; e.g., the size of index file is fixed.

3. *dynamic SPIN* means the SPIN package developped by the SPIN research group[1]. There are several dynamic SPIN algorithms proposed so far. In this thesis, two dynamic SPIN schemes are presented: index dynamic SPIN (*D_SPIN* for short), and directory dynamic SPIN (*T_SPIN* for short). The size of index file is flexible in dynamic SPIN schemes.

[1]This research group consists of Dr. G. E. Hedrick, Dr. R. A. DiVall, M. Z. Fan, and Y. Zhang.

4. *load factor* means a number, λ, whose value is between 0 and 1, and whose value indicates how full (the load) the storage tree (or table) is. An empty tree (table) has a load factor of 0 (λ=0); a full tree (table) has a load factor of 1 (λ=1).

5. In SPIN, *overflow* means an attempt to insert data into a tree with a load factor of 1. A memory error occurs if *overflow* is not handled properly.

6. *radix[tree]* means a tree that has a fixed number of [possibly empty] branches from each node. A radix n tree has n branches at each node.

7. A *bucket (or page)* corresponds to one or several physical sectors of a secondary storage device such as a disk. The capacity of a bucket is b records in this thesis.

8. *Space utilization* is the ratio between n and $m*b$, where n is the number of records in the file, m is the number of pages used, and b is the capacity in records of the page.

CHAPTER III

STATIC SPIN SCHEMES

Coburn[4] developed three basic SPIN functions: *C_SPIN*, *S_SPIN* and *R_SPIN*. All three functions support multidimensional data spaces and have the static property. The basic mechanism behind these functions is that a key is transformed into an index value. The index value is used to find an address of records in the data file. SPIN uses a series of algorithms derived from the Fundamental Principle of Counting(FPC). The FPC states:

> Given a series of m operations 1, 2, 3, ..., m. If the first operation can be performed in m_1 ways, the second in m_2 ways, and so on until the mth operation, which can be performed in m_n ways, then the number of ways the m operations can be performed is

$$\Pi_i =_{0,n} m_i \tag{1}$$

The C_SPIN Function

Coburn uses a multidimensional array to represent a multidimensional data space. For instance, in the C language, the declaration "int arr[3][3][3]" represents an array of 27 integers in three dimensions with three indices in each dimension. When using *C_SPIN*, this array represents the three leveled spaces, corresponding to three dimensions of the array.

The problem addressed by *C_SPIN* is: "How can one access the multidimensional array?" Traditionally, the programmer uses cursors which are translated to pointer

variables to navigate through the dimensions of a multidimensional array. This makes partial key search, storage, and retrieval operations impossible. Also, procedures for allocating multidimensional memory dynamically are generally unavailable.

One solution is to convert the multidimensional array into a corresponding single dimensional array. With a one dimensional array, the shortcomings of accessing a multidimensional array with a pointer variable can be avoided. Another reason for using the single dimensional array is that the digital computer is a sequential device, but multidimensional arrays are not structured sequentially. One dimensional arrays are sequential. Coburn states that nonsequential data structures can pose special storage and retrieval difficulties for the computers[3].

Given the array "arr[3][3][3]", it follows from formula 1 that there are 27 valid multidimensional subscript combinations(keys) for this array. In general, given an N dimensional array with n_i indices in dimension i and $i = 0, 1, 2, ..., N-1$, by formula 1, *C_SPIN* transforms the $n_0 * n_1 * n_2 * ... * n_{N-1}$ subscript combinations into the set of sequential whole numbers k such that $k = 0, 1, 2, ..., n_0 * n_1 * n_2 * ... * (n_{N-1} - 1)$. This transformation has the following properties[4]:

1. The transformation is well ordered. That is, given two distinct subscript combinations(keys), say k_1 and k_2, which are passed to the *C_SPIN* function with the proper arguments, and two return values O_1 and O_2 from *C_SPIN* corresponding to k_1 and k_2 respectively. it follows that if $k_1 > k_2$, then $O_1 > O_2$.

2. The transformation is one-to-one and onto. For each distinct subscript combination passed to *C_SPIN*, there is exactly one return value and that return value is unique.

3. The transformation is sequential. A given multidimensional array contains keys that are base 10 integer values, it follows that, given k_1 and k_2 such that $k_1 < k_2$, if there does not exist an k_k such that $k_1 < k_k < k_2$, then the corresponding

[0][0][0] <—> 0	[1][0][0] <—> 4
[0][0][1] <—> 1	[1][0][1] <—> 5
[0][1][0] <—> 2	[1][1][0] <—> 6
[0][1][1] <—> 3	[1][1][1] <—> 7

Figure 1: Transformation between multidimensional array and single dimensional array

return values O_1 and O_2 have the following relationship: $O_2 = O_1 + 1$.

The Figure 1 clearly shows the above properties, given a multidimensional array "int arr[2][2][2]".

While *C_SPIN* provides a good way to access multidimensional arrays in main storage, there are still some problems with *C_SPIN*. The first problem is that *C_SPIN* is restricted to dimension sizes no larger than five. The reason is the formula that *C_SPIN* employs to conversion grows exponentially when the number of dimensions increases. The second problem is that access is denied to the interior dimensions of a multidimensional array structure, therefore, partial subscript combinations cannot be indexed. These limitations lead to the *S_SPIN* function.

The S_SPIN Function

S_SPIN function iteratively applies *C_SPIN* function one dimension at a time. The formula *S_SPIN* uses in transforming a multidimensional array into a singly dimensional array is:

$$rec_number[i+1] = rec_number[i] * 10^{ex[i]} + k[i+1] - rec_number[i] * kr[i] \quad (2)$$

where,

rec_number[i+1]=the index for the (i+1)st level.[2]

rec_number[i]=the index for the previous level or iteration.

max[i]=the number of indices in the ith dimension.

ex[i]s are exponents computed as followings:

if(max[i+1]>0 && max[i+1]<=10) ex[i]=1

else if(max[i+1]>10 && max[i+1]<=100) ex[i]=2

else if(max[i+1]>100 && max[i+1]<=1000) ex[i]=3

......

k[i]=the value of the multidimensional subscript within the ith dimension.

kr[i]s are values computed as followings:

if(max[i+1]>0 && max[i+1]<=10) kr[i]=10-max[i+1]

else if(max[i+1]>10 && max[i+1]<=100) kr[i]=100-max[i+1]

else if(max[i+1]>100 && max[i+1]<=1000) kr[i]=1000-max[i+1]

......

$i \subset (0, 1, 2, 3, ..., N - 2)$;

As an example, the following illustration using the array "arr[2][4]" applies the formula 2:

N=2;

max[0]=2, max[1]=4;

ex[0]=1, since max[1]<=10;

kr[0]=10-max[1]=10-4=6.

Eight valid subscript combinations can be passed to *S_SPIN*. They are (0,0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3). By formula 2, $rec_number[1] = rec_number[0] * 10^1 + k[1] - rec_number[0] * kr[0]$; where, rec_number[0] is the index

[2]One less than the number of the dimension within a multidimensional array is called the level of the array.

Subscript Combination	Right-hand side computation	Index
(0, 0)	$0 * 10^1 + 0 - 0 * 6$	0
(0, 1)	$0 * 10^1 + 1 - 0 * 6$	1
(0, 2)	$0 * 10^1 + 2 - 0 * 6$	2
(0, 3)	$0 * 10^1 + 3 - 0 * 6$	3
(1, 0)	$1 * 10^1 + 0 - 1 * 6$	4
(1, 1)	$1 * 10^1 + 1 - 1 * 6$	5
(1, 2)	$1 * 10^1 + 2 - 1 * 6$	6
(1, 3)	$1 * 10^1 + 3 - 1 * 6$	7

Figure 2: Transformation values for the array arr[2][4]

for the first dimension, it is zero or one in this example. k[1] is the index value in the first level, it is 0, or 1, or 2 in the example. Figure 2 shows the transformation.

The formula 2, a recursive function, generates a sequence of return values at each level. For example, given an array "arr[1][2][3][4]," when applying formula 2 to calculate the index value for this array, it also generates index value for "arr[1][2]," and "arr[1][2][3]," as well as "arr[1][2][3][4]." Consequently, *S_SPIN* permits computation of partial subscript combinations (partial keys). There are some limitations of *S_SPIN* One, is that the level order is fixed. The level order cannot be changed without restarting the entire computation. The other problem is, since the level order is fixed, accessing level n requires computing values for all levels through and including level n-*1* first. It is not possible to access the level we want directly. Partial subscript combinations always proceeds from the first level to the desired level; It is uni-directional. For example, given the array "arr[1][2][3][4][5]," we cannot access a partial subscript combination such as "[4][5]," "[2][3][4]." In a relational database

environment, the relationship reflected by such combinations is common.

The R_SPIN Function

Another problem *C_SPIN* and *S_SPIN* cannot solve is representation of sparse data. Database designers try to avoid the problem of representation of sparse data. For example, suppose an airline owns 100 planes and flies to 100 cities. It is reasonable to assume that the route that each plane flies does not pass through all 100 cities. A multidimensional array to represent this would be "arr[100][100]." Using such an array would require using 10000 (100*100) bytes of storage. If each plane flies to at most 3 cities on a route, we would like to declare the array "arr[100][3]," because it would require using only 300 bytes of storage, a savings of 9700 bytes.

R_SPIN can keep track of which nodes at one level are mapped to which nodes of the next level. This helps to solve the problem of sparse data representation. Given a multidimensional array, such as "p[20][40][50]," first, we assume we know that: a) whether there is a sparse situation; and b) the degree of sparseness. Suppose, after eliminating sparseness, we can reduce the number of mappings in the "p" array to "p[20][3][2]," which is a dense array. We open an index file constructing a tree structure for "p[20][3][2]" (Figure 3).

We construct a one-to-many mapping. From "p[20][3][2]" each element in level zero is mapped to three elements in level one; each element in level one is mapped to two elements in level two(Figure 3). There are 20 trees in Figure 3, corresponding to the first dimension in "p[20][3][2]" which has 20 elements in dimension one. At the beginning, all nodes are initialized to zero except those at level zero, into which are put into index values taken from the first dimension of "p[20][3][2]" (Figure 3). If *R_SPIN* operates on the subscript combination "p[3][20[8]," *R_SPIN* first uses formula 2 to calculate the index values corresponding to "p[3][24]" and "p[3][24][8]," then *R_SPIN* stores the index values in the first empty node at the corresponding level. We call

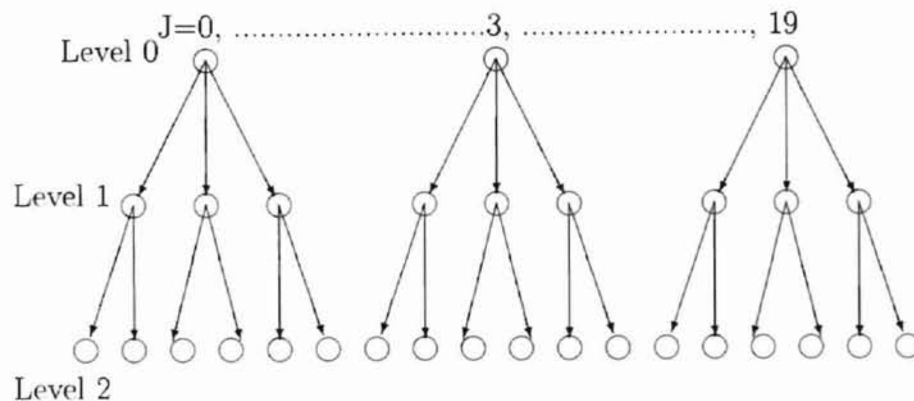


Figure 3: Tree structures for R_SPIN

these values V values.[3]

Next, *R_SPIN* uses the location value of the nodes to calculate the index values in the dense array by employing formula 2. For example, if V values of the array "p[3][24][8]" are stored in the first node at each level, then the subscript combination of location values is "k[3][0][0]."

R_SPIN retrieves index values for a subscript combination by matching the value computed by formula 2 with those stored in the index file. For example, if the array "p[3][14][8]" is passed to *R_SPIN*, it will employ formula 2 to calculate level one index value using "3" and "14" as subscripts. Then it will compare the V value stored at each one of the three nodes of level one of the tree which is indexed 3. If it matches one of those values, it will go to next level to determine whether the index values calculated using "3", "14" and "8" matches the V values stored at that level. *R_SPIN* significantly reduces the number of mappings into sparse array. This creates a very efficient way of storing and retrieving keys.

The limitations of R_SPIN

The major deficiency with *R_SPIN* is the inability to predict accurately the sparseness

[3]The first empty node at each level means the first available node at certain subtree of that level, rather than all nodes of that level.

OKLAHOMA STATE UNIVE

arbitrary multidimensional arrays. Failure to make such predictions accurately leads to an overflow. The following example shows the overflow problem.

An airline example

For example, suppose we have a level-structured airline network with the airline routes emanating from a:

a. very big city;

b. big city;

c. middle-sized city; or a

d. small city.

We use a multidimensional array to express this structure as: $P[a][b][c][d]$. Suppose, we have an airline from a to b, b to c, and c to d. At level two going from b to c, we may have such questions as: From a given big city; e.g., Dallas, how many airline routes are there to middle-sized cities? This leads to the "distribution of keys" problem.

Distribute of keys

A statement of the "distribution of keys" problem follows. On average, how many airlines are there from one of the big cities to the middle-sized cities? We must know the probability of Dallas (a big city) having five or more airlines to the next level (middle-sized cities). We also must know what the probability of one big city having five or more airlines to the next level (middle-sized cities). We examine several sample data sets, then determine the distribution of keys for these data sets. For example, we can choose Houston, Chicago, ..., etc., to determine whether each of these cities has five or more airlines to the next level. We compute an average probability for each of these cities having five or more airlines each to the next level of cities. If the samples are typical and representative, then we apply the result to the entire data set even though the contents of the data set change dynamically.

From this example, we surmise:

1. The distribution of keys can be predicted only in a detail application environment.

2. The number of actual mappings assigned to a certain node at a certain level of some leveled data structure depends on the prediction for the distribution of keys. For example, if Dallas has a .95 probability of having five airlines fly to the next level, and we assign only four airlines to Dallas, then there is a very high risk of an overflow situation .

3. The probability differences for overflow exist not only among different levels, but also among different nodes at the same level. For example, suppose Houston is at the same level as Dallas. Houston's probability of having five airlines fly to the next level might be .85. One reason is that the actual number of airlines for Houston and Dallas might be different. Dallas might have six airlines; Houston might have eight airlines.

4. In a detailed application environment, the overflow problem may be predicted, but the problem still exists. Even with a .99 probability of having no overflow for a certain node, the risk of overflow is still there. For example, suppose Dallas has .95 probability for five airlines, .99 probability for six airlines. If we assign six airlines to Dallas, there is still a .01 probability of overflow. One solution is to assign more positions for airlines to Dallas. In such case, the risk of overflow may approach zero, but space is wasted in a matrix representation of the data. In other words, Dallas uses only a small number of the airlines assigned. Complete elimination of inefficient (sparse) data storage is *R_SPIN*'s target, but that also causes the overflow problem. It is a dilemma: complete elimination of sparseness causes overflow; complete elimination of overflow requires sparse data to be

OKLAHOMA STATE UNIVE

stored in oversized arrays.

5. R_SPIN employs formula 2 twice. First, it uses formula 2 to calculate an index value. Since formula 2 has an estimated time complexity of

$$O = (c^n)$$

for some machine defined constant, c, and number of dimensions, n, its repeated use during a single operation makes the method relatively inefficient.

6. The sparse situation that occurs in this example is: there are many middle-sized cities, but not all of them are connected to all of the big cities.

OKLAHOMA STATE UNIVE

CHAPTER IV

DYNAMIC SPIN SCHEMES

In this chapter, we introduce two dynamic SPIN schemes: index dynamic SPIN (*D_SPIN* for short) and directory dynamic SPIN (*T_SPIN* for short). Both algorithms evolved from the limitations of the static SPINs. We explore some design issues, such as space utilization, splitting control, and bucket structure. We also compare the dynamic SPINs with the static SPINs.

Index Dynamic SPIN

From the limitations of the *R_SPIN*, we design a new index SPIN, which dynamically changes its number of nodes when overflow happens. It eliminates the dilemma presented in Chapter III.

Preliminaries

Mappings between levels

The example in last chapter shows that different nodes at the same level may have different numbers of mappings, since the nodes may have different probabilities of overflow. The trees in *R_SPIN*'s index file are not the same. A level may look like those shown in Figure 4. In Figure 4, tree one has three children at level one and each child at level one has two children at level two. For tree two, there are two children at level one - one has two children at level two, the other has three children at level two.

OKLAHOMA STATE UNIVERSITY

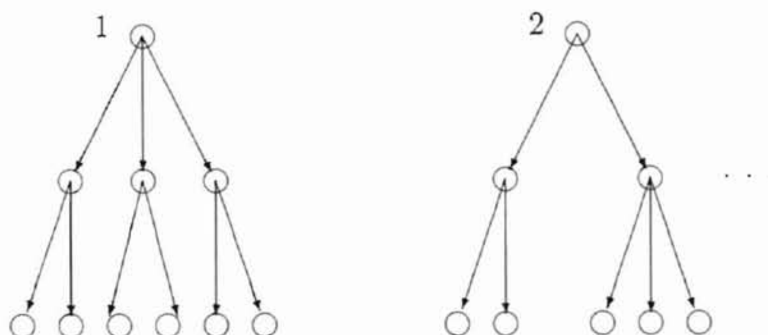


Figure 4: Different shapes of trees in an index file

Sparse data mappings

It is common in sparse storage situations with *R_SPIN* to have different numbers of mappings at the same level. Suppose a teacher teaches three classes, each class has a maximum 30 students. A multidimensional array to express the teacher's classes with the students in each class requires a multidimensional array, V[3][30]. However, if only 10 students enrolled in the first class, 20 students enrolled in the second class and 27 students enrolled in the third class, then there is a sparse data storage situation. Each number at the first level (0, 1, 2) has a different mapping to the second level (10, 20, 27).

Even when there is the same number of mappings for each node at the same level, the nodes commonly have different contents. The 30 students in class one will be different from those in class two, as well as from those in class three. There exist situations that we have the same contents for multiple mappings. For example, some students may enroll in more than one class. Taking the above information together, a level-structured graph shows the sparse storage situation clearly.

Figure 5 shows four nodes at level one and nine nodes at level two. Each node at level one may have 1, 2 or 3 mappings to level two. Some mappings have the same

OKLAHOMA STATE UNIVERSITY

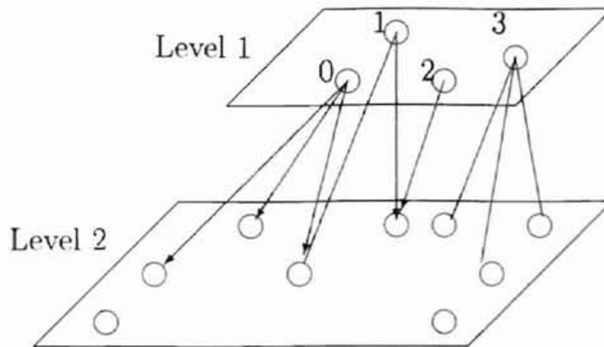


Figure 5: A leveled structure for the sparse situation

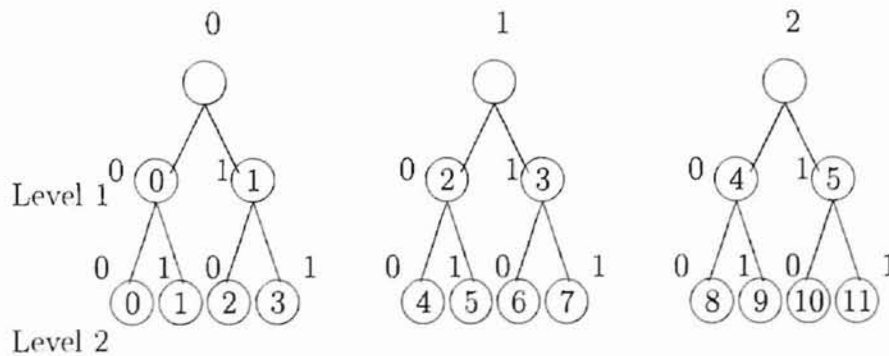


Figure 6: Sequential natural numbers indexing the nodes at the same level

content (Two or more pointers at level one point to the same node at level two). Also, some nodes at level two are unused.

Tree structured arrays

Figure 6 shows a tree structure corresponding to the multidimensional array $A[3][2][2]$. The subscript key values appear beside the nodes.

Formula 2 (Chapter III) calculates the index values at level one. It results in:

```
[0][0]=0  [1][1]=3
[0][1]=1  [2][0]=4
```

OKLAHOMA STATE UNIVERSITY

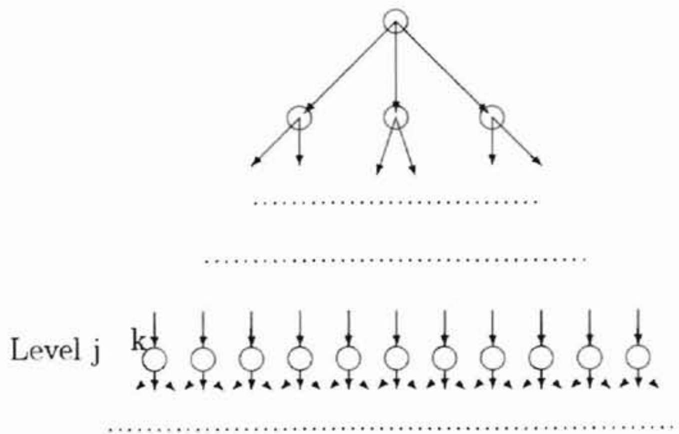


Figure 7: A no-sparse (dense) tree

`[1][0]=2 [2][1]=5`

These index values correspond to nodes as shown in Figure 6.

Level one index values are sequential numbers progressing from the leftmost node to the rightmost node. The index values for level two also may be sequential numbers. Index values fill the nodes (Figure 6).

Formula 2 yields:

```
level two
[0][0][0]=0  [1][1][0]=6
[0][0][1]=1  [1][1][1]=7
[0][1][0]=2  [2][0][0]=8
[0][1][1]=3  [2][0][1]=9
[1][0][0]=4  [2][1][0]=10
[1][0][1]=5  [2][1][1]=11
```

Formula 2 and Figure 6 are consistent. Both reflect the transformation properties, which are stated in Chapter III.

Let the tree i, (Figure 7) be dense. At level j, the first node's index value is k, the next node will be k+1, next to next: k+1+1, ..., until the last node (rightmost

OKLAHOMA STATE UNIVERSITY

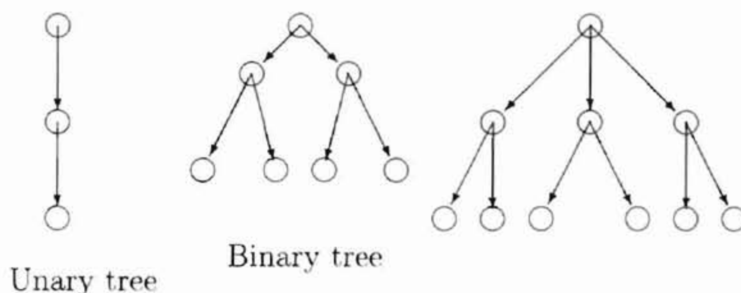


Figure 8: Different initial trees

node) at level j. The values appear as:

$k, k+1, k+2, k+3, k+4, ...$

We also know that the index values for tree i-1, which is at the left side of tree i, at level j. The last node's index value at level j for tree i-1 is *k-1*. Continuing to the left, we have all index values at level j:

$....k-4, k-3, k-2, k-1.$

General description

Given a multidimensional array (or combination of keys), $P[20][30][40]$, we determine the maximum number of actual mappings at each level to eliminates sparseness. This does not eliminate overflow. In other words, we may choose $p[20][1][1]$, which certainly eliminate sparseness, but also has an overflow problem. Overflow is not a problem during initialization. Figure 8 shows several structural choices for eliminating sparse data storage.

If it is unknown whether there is a sparse storage problem, or if it is known there is a sparse storage problem and the probability of overflow is unpredictable, then, the unary tree may be the best choice to guarantee dense storage utilization. Binary trees

OKLAHOMA STATE UNIVERSITY

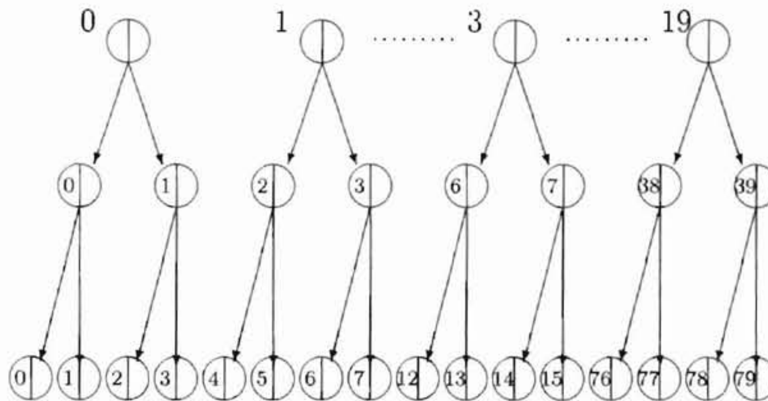


Figure 9: Trees with splitted nodes

also can be used in certain applications.[4] If we can predict the probabilities, we may choose a conservative number of mappings between levels to minimizing the risk of overflow, while we also eliminate sparse data storage problems.

The reason for choosing among different shapes of trees is to initialize those trees efficiently. Any tree will grow after initialization. A tree growing from a binary structure allows faster access than a tree growing from unary structure. There is no significant increase in access speed when using a tree with radix greater than two.

The binary tree is the starting tree in this example; i.e., $p[20][2][2]$. Each node splits into two parts, shown in Figure 9.

Using the preliminaries described above, we fill in the index values the left part of the node, shown in Figure 9. At the same level, indexes are sequential numbers. The index value for the root node is trivial, since the number in dimension one is always the same after eliminating spars storage; i.e., from $P[20][30][40]$ to $p[20][2][2]$, the number 20 does not change.

Given a level two datum with key k_1 and with index [3][14][8] ($k_1[3][14][8]$), we employ formula 2 to calculate the V values for k_1. Suppose V_1 is the V value for

[4]A binary tree can be used in any application. It is particularly useful in applications such as the airline problem.

OKLAHOMA STATE UNIVE

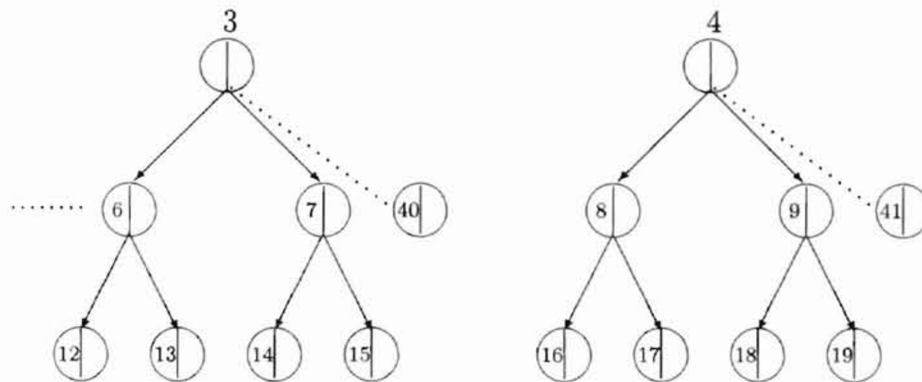


Figure 10: Insertion situation for index dynamic SPIN

level one, $V_1 = k_1[3][14]$. We put V_1 into the right part of the first available node at level one, shown in Figure 9. The index value for V_1 is just the number in the left part of node. In this case, it is six.[5] Let V_{11} stands for the V value for level two, $V_{11} = k_1[3][14][8]$. We put V_{11} in the first available child node of V_1. We also can obtain the index value for V_{11} easily. In this case, it is twelve.

Suppose we want to insert $k_2[3][17][21]$. V values(V_2, V_{21}) will be put in the corresponding nodes, shown in Figure 9. Next, suppose we insert $k_3[3][19][7]$. Since there is no empty node at level one available, there is an overflow situation.

To handle the overflow situation, we create an empty node at level one, which is also connected to the root node 3 (Figure 10).

The newly created node is split into two parts as before. The index value for the new node is the maximum index value at level one (which is 39) plus one; i.e., 39+1=40. We put number 40 in the left part of the new node. The right part of new node is still used to store V values. All other nodes are left unchanged. Now the maximum index value for level one is 40 (Figure 10). If tree 4 has overflow at level one, then we create a new node, split it, put the index value 41 (40+1) into the

[5]we do not need to use Formula 2 to calculate index values. The index values are already in place.

OKLAHOMA STATE UNIVERSITY

left part of the node, and put the V value into the right part of the node (Figure 10). Index values for level one are no longer sequential numbers. However, this new method observes a prime principle of SPIN: a distinct V value is associated with a distinct index value. The index values are used to retrieve data records from a database.

We discuss several common operations for this new algorithm.

1. Insertion

   Insertion is discussed above. An inserted node is filled with two numbers. The left part of the node is its index value, the right part of node is its V value. Also, the left part of each node should never be empty. An index value should always occupy the lefthand "data area."

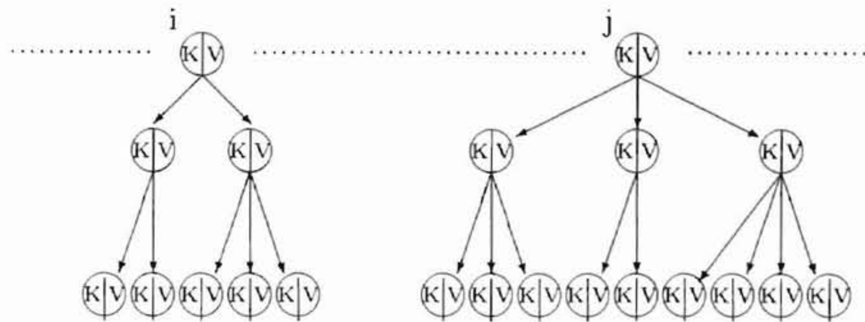
2. Searching

   This uses the *R_SPIN* method for searching. It calculates V values for the key, or combination of keys, then uses the calculated V values to search the corresponding tree, node by node, level by level. If the calculated V value matches the V value stored in the node, then search is successful. The index value of the node is returned. The search is unsuccessful when a V value of 0 is deleted.

3. Deletion

   Logically, if the deleted node's index value is X, then we must find all nodes whose index value is larger than X, then reduce their values by one. A problem arises with those nodes whose index values are larger than X. They can be located randomly on both sides of the deleted node. We must search every node at that level, and check whether the index value is larger than X. One way to avoid this tedious work is lazy deletion. When we want to delete a node, we

OKLAHOMA STATE UNIVERSITY



K: index value;

V: V value;

Figure 11: The final result of the index tree in index dynamic SPIN

mark it as deleted, but leave it physically in place. When the marked nodes reach a threshold, we start the physical deletion mechanism (garbage collection) –physically deleting all marked nodes during a single search.

After building these trees, a possible final result is shown in Figure 11. The shape of each tree is different.

This new method has several advantages:

1. It completely eliminate sparse data storage problems
2. It eliminates overflow.
3. It employs formula 2 only once.

Like most traditional tree algorithms, all trees' shapes are changing (growing or shrinking) dynamically, according to the operations performed.

Space Utilization

Assume that data is stored on magnetic disks. Data is fetched from magnetic disk drives in pages of a fixed size. Further assume we have built a very large index file of more than 100,000 trees on that disk already. Each fetch of a page takes about

OKLAHOMA STATE UNIVERSITY

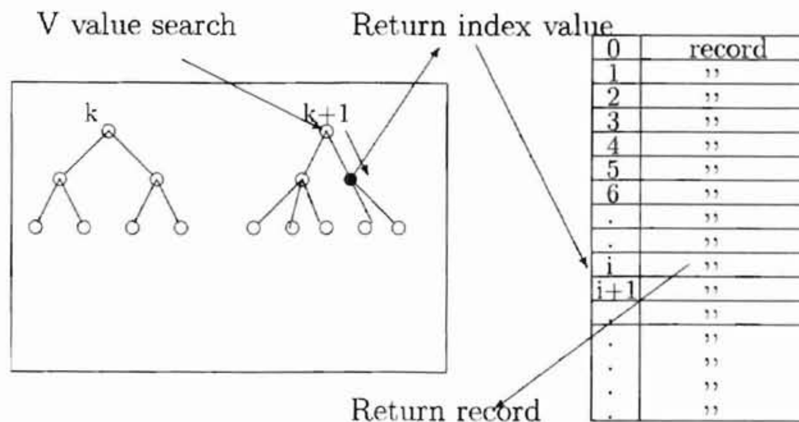


Figure 12: The whole picture for data indexing and retrieving

10 milliseconds on average on the fastest disk drives. This is the time it takes for the disk arm to move to the correct cylinder, for the disk to revolve until the head is over the correct place in the track, and for the data to be transferred to the main storage. For a search operation, given a combination of keys, index dynamic SPIN scheme calculates a V value, then the program accesses the index file to try to find the same V value. The index file consists of several pages (blocks). The page that contains the tree number is brought into main storage.

Suppose each page contains 30 trees. Since the root numbers of trees are kept as sequential numbers, retrieval of the tree with the index value matching the V value is direct. The index value points to a location in a data file which is also on the disk and broken into pages by the operating system[6]. What is associated with each index value may be a data record, or another file pointer, depending on the application. Page fetch or swapping is the same as above. The total situation is shown in Figure 12.

No matter how large a page is, the root numbers of the tree are sequential natural numbers, as are the index values. This kind of structure is useful since a

[6]All index values are arranged into sequential numbers in all pages.

OKLAHOMA STATE UNIVERSITY

digital computer is a sequential device.

Directory Dynamic SPIN

First we presented preliminaries of directory dynamic SPIN (*T_SPIN* for short). Then we gave a general description and several design issues about *T_SPIN*.

Preliminaries

A General Design Method for SPIN

The transformation mechanism behind *R_SPIN* is studied first. Given a sparse multidimensional array, such as P[20][30][40], and its dense form p[20][3][2](different from *D_SPIN*, *R_SPIN* gives us both a sparse form and a dense form before we can go through the *R_SPIN* function), we have an index file containing 20 tree structures of the form shown in Figure 3.3. All nodes initially contain the value zero. When the subscript combination, such as P[3][24][8], is passed to *R_SPIN*, *R_SPIN* uses the formula 2 to calculate the corresponding index value, such as V. This is the first transformation, which transforms p[3][24][8](a multidimensional data space) to integer number V (a singly dimensional data space). Such transformation is not enough, since the result V is in the sparse data space. We need to transform V (big number, sparse form) into n (small number, dense form). In order to do that, we stores the V in the first empty node at the corresponding level. In this case, it's level 2. Then, *R_SPIN* uses the location value of that node into which it already put V , to calculate the dense form index value n using the formula 2. *R_SPIN* returns the value n.

We can see from *R_SPIN* that the whole process actually is divided into two steps:

First step, the transformation from a multidimensional data space to a single dimensional data space.

OKLAHOMA STATE UNIVERSITY

Second step, the transformation from a sparse data space, which is already single dimensional, to a dense data space.

D_SPIN also shows this two-step transformation. First step, *D_SPIN* is the same as *R_SPIN*; Second step, because *D_SPIN* already stored the index value(n value in this example) in the left part of node, it retrieves the n value from the node, instead of calculating it. So, if we work in a sparse multidimensional data space, the above two-step transformation is necessary if we want to index and process data efficiently. Besides *R_SPIN* and *D_SPIN*, various techniques can be used in the process of transformation. All those techniques either have a static property, as does *R_SPIN*, or a dynamic property, as does *D_SPIN*. In the view of design, the two-step transformation gives us a general design method to design a new SPIN technique. In other words, we only need to design a transformation technique for each transformation step. The *T_SPIN* proposed in this section is designed according to the two-step transformation design method.

General Description

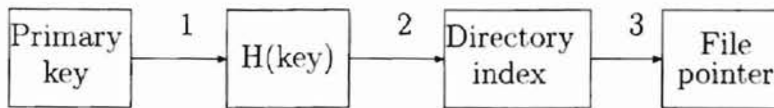
T_SPIN is evolved from extensible hashing[8]. Extensible hashing is a method of organizing a file so that it has the following three properties[13]:

1. The file will automatically expand as necessary to accommodate new records. The expansion will not require a reorganization of the file.

2. The file will automatically contract so that the probability of the load factor dropping below 50% is negligibly small. As with the expansion, the contraction will not require a reorganization of the file.

3. The file structure will allow retrieval of a record by primary key with one access to the file.

OKLAHOMA STATE UNIVERSITY



1: Hashing function(S_SPIN function in directory dynamic SPIN)

2: Extract first d digits

3: Table look up

Figure 13: Key transformations in extensible hashing

Obviously, extensible hashing file is a dynamic file structure. The sequence of transformations by extensible hashing is shown in Figure 13. The first transformation is a hashing function that maps the keys randomly onto some fixed address space represented by the range of the hashing function. The first few digits of this result are then extracted for use as an index into a directory. The directory contains pointers that point to the file.

Considering the two-step transformation method in SPIN design, we can see that the first transformation in Figure 13 actually is doing the work for the second step in the two-step transformation: transforming a single dimensional sparse data space into a dense data space.

In order to combine the dynamic properties of extensible hashing with SPIN techniques, we must add the first step of two-step transformation to Figure 13, which is, a transformation from a multidimensional data space to a singly dimensional data space. We employ formula 2 to do this work. However, we do not need any tree structures in this step, because the results of first step are arranged linearly so that they can be easily hashed in the second step transformation.

Now, the whole picture of transformations is shown in Figure 14.

OKLAHOMA STATE UNIVERSITY

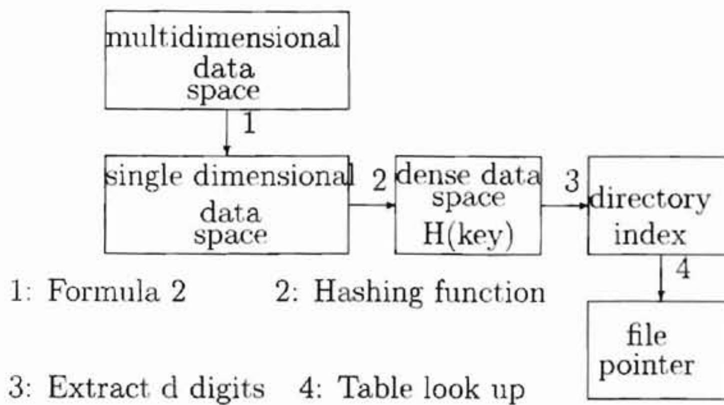


Figure 14: Key transformations in T_SPIN

There are several design issues related to the transformations in Figure 14.

1. In Figure 4.11, only first three steps transformation belongs to *T_SPIN*, because the fourth step (table look up) is beyond a indexing technique, and it will be handled by file system.

2. The file pointers do not point to individual records, like *R_SPIN* or *D_SPIN*, but rather to blocks of records called buckets. A bucket is a large block of records, all of which are read with one physical read operation. The method of placing and locating records within the bucket is not important since no additional physical I/O operations are required[23]. Buckets may be added to and deleted from the file at any time.

3. The hashing function used with *T_SPIN* can be any hashing function as long as it satisfies the following four properties[13]:

   (a) uniform and random distribution of keys over the range of the function;

   (b) small variations in the key will cause large variations in the value of the function;

   (c) synonyms occur no more than random probabilities would allow;

OKLAHOMA STATE UNIVERSITY

(d) the range of the hashing function be close to a power of 2.

4. The selection of the range of the hashing function is somewhat arbitrary and is not tied to the number of records in the file. The range of hashing function is the range of address space of dense data. For a dynamic scheme, we cannot know in advance precisely how large the dense space need to be[4]. The selection of the range of the hashing function actually depends on the given practical application. In *R_SPIN*, such arbitrary selection of the range of dense data space leads to the overflow problem; in *D_SPIN*, there is no this problem, because the index trees (which actually reflects the range of dense data space) dynamically expands or contracts. In this case, *T_SPIN*, the arbitrary selection may lead to synonyms, which means more than one record is mapped to the same location by the hashing function. Considering the basic unit in a data file is the bucket in *T_SPIN*, if synonyms happen, we can put two records in the same bucket. This means we map them to the same location.

5. According to the properties of the hashing function, it is convenient to choose a range of hashing function equal to the first prime number smaller than a "round" binary number that is a power of 2[7]. For example, the largest prime number less than 2^{16} is 65,521.

6. The third transformation in Figure 15 extracts a relatively small integer from H(key) by using the first few digits of the hashing function. There are several arbitrary choices in the selection of digits. It is advantageous to use as small a base as possible; hence, binary will be used as the base and bits will be the digits. Another choice is which digits to use. Conventionally, the high-order digits are selected[7].

7. The digits extracted from H(key) are then used as an index into a one dimen-

OKLAHOMA STATE UNIVERSITY

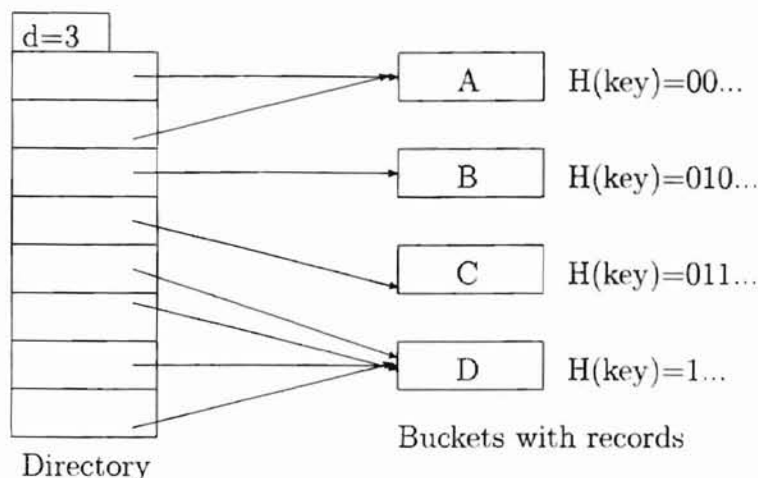


Figure 15: Directory of order d=3 with four buckets

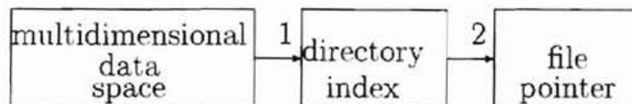
sional array of file pointers. This array is called the directory and contains 2^d entries, one for each combination of d digits from H(key).

8. The number of digits extracted from the hashing function value, the number of entries in the directory, and the number of buckets in the file all will change automatically as the file expands and contracts. Consequently, it is necessary to store some parameters to indicate the current state of the file. Specifically, the number of digits, d, used to index into the directory are stored with the directory as shown in Figure 16.

The transformation shown in Figure 16 uses the first three binary digits from the hashing function to partition the address space of the hashing function into eight equal segments. These eight segments correspond to eight entries in the directory. For example, suppose H(key)=0110100101100101 in binary. The first three digits, 011, have a value of 3. By using 3 as an index into the directory, we find a pointer that points to bucket C.(The first element in the directory has an index of zero.)

The complete sequence of retrieving a record using *T_SPIN* consists of six steps. First, the formula 2 is applied to produce a single dimensional data(key). Second, the key is hashed to produce H(key). Third, the first d digits(bits) are extracted from

OKLAHOMA STATE UNIVERSITY



1: The T_SPIN function

2: Table look up

Figure 16: *T_SPIN* transformations

H(key) to form an index into the directory. Forth, the index is used to locate the appropriate bucket pointer in the directory. Fifth, the pointer is used to read the bucket into primary memory. Sixth, the desired record is located within the bucket.

The *T_SPIN* Function

From Figure 15, we see that the first three steps of key transformation are actually functional operations. The output of formula 2 is the input of the hashing function, and the output of the hashing function is the input of the extraction function. This chain of function calls suggests that we may combine the three functions into one function to eliminate the overhead of function calls. The *T_SPIN* function is the result of such combination. The input of the *T_SPIN* function is the multidimensional array and the number of digits we want to draw (d value), and output of the *T_SPIN* function is the digits extracted from the hashing result (the hashing process is embedded in the *T_SPIN* function).

After combining the function calls together, the key transformation is simplified as shown in Figure 17.

Splitting Control

OKLAHOMA STATE UNIVERSITY

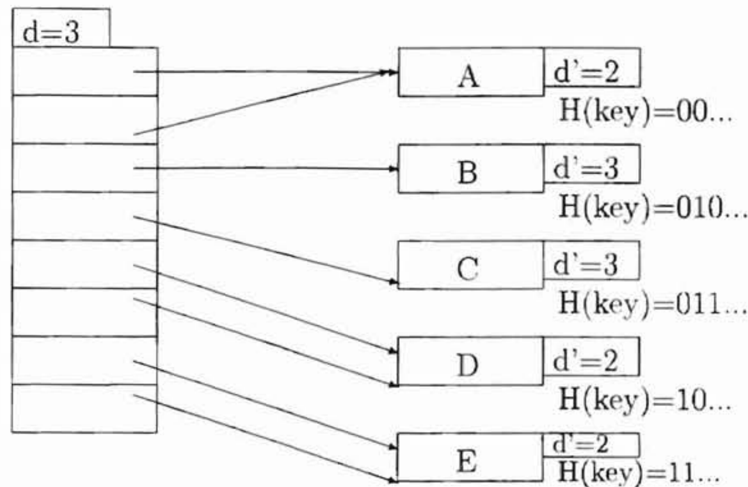


Figure 17: Distribution of keys among buckets after splitting bucket D.

The reason we use buckets instead of records as basic units in data file is to allow the data file to expand or contract gracefully as the number of records varies. In other words, we want to keep the perfect dynamic property for *T_SPIN*. In this section, we discuss the splitting control in data file.

A rule is imposed on the buckets that sets a minimum and maximum load factor for each bucket. Typically, these are 50% and 100%, respectively[13]. A change in the file structure is triggered whenever these limits are violated by the addition or deletion of a record.

Consider the small file of Figure 16. Suppose that a record is to be added which maps into bucket D. If bucket D is already full, there is no room for the new record. This triggers an expansion of the file. A new bucket, E, is added to the file. Half of the pointers that point to bucket D are changed to point to bucket E. The records in bucket D that are reached through the pointers that were changed must be moved to bucket E. This will be approximately half of the records that were in bucket D. This leaves both buckets approximately half full and there is ample room for the new record.

The result of this split is shown in Figure 17. Before the split, all records for

OKLAHOMA STATE UNIVERSITY

which H(key)=1... were in bucket D. Now those where H(key)=10... are in bucket D and those where H(key)=11... are in bucket E. The parameter d' shown with each bucket in Figure 17 indicates the number of digits of H(key) whose value is common to all records in the bucket. This must always be equal to or less than the number of digits, d, used to index into the directory.

The process for contracting the file is the reverse of enlarging it. Buckets must be combined if three conditions are true[9]. First, the average load factor for the two buckets cannot exceed 50%. Otherwise, there would not be room in the combined bucket for all the records. Second, the buckets to be combined must have the same value of d'. Third, the keys of the records in both buckets must share a common value of the first (d'-1) digits of H(key). These last two conditions are necessary so that the records of the combined bucket will share a common value of the first d' digits of H(key).

OKLAHOMA STATE UNIVERSITY

<u>Bucket Structure</u>

Besides the splitting control, another important design issue is the bucket structure. However, we don't need to pay much attention to bucket structure for two reasons. First, whatever method is used to organize the bucket internally will not affect the number of physical I/O operations and so will not have a significant impact on most file operations. Second, there are many feasible solutions, with no clear preference between them[7].

Structures as simple as a sequential-chronological organization with a bit map are feasible. To find the desired record, each record in the bucket is examined in sequence and its key is compared to the given key until a match is found or the end of the bucket is reached. The ordered relative file could also be used for the internal bucket structure and would permit a binary search to be used to find the desired record. The problem of insertions and deletions is solved by moving blocks of records

as necessary to make room for a new record or close up a gap when an old record is removed. Thus, the reorganization is continuous and can be done entirely within main storage.

Comparisons Between Dynamic SPINs And R_SPIN

The comparisons between dynamic SPINs and *R_SPIN* in this section focus on the design issues. The performance comparisons are covered in Chapter VI.

There are several differences between dynamic SPINs and *R_SPIN*:

1. *T_SPIN*, *D_SPIN* and *R_SPIN* can all be analysed by two-step transformation method. But they have different operations on second step.

SPIN	First step	Second step
R_SPIN	Formula 2	Formula 2
D_SPIN	Formula 2	embedded in the nodes
T_SPIN	Formula 2	hashing function

2. The contents of index file and basic units of data file are different among the three SPINs.

SPIN	Contents of index file	Basic units of data file
R_SPIN	tree structures	records
D_SPIN	tree structures	records
T_SPIN	directory table	buckets

3. Different overflow problem handling techniques:

   (a) *R_SPIN*: gives a message when overflow happens;

   (b) *D_SPIN*: solves the overflow problem by dynamically expanding or contracting the tree structures index file;

   (c) *T_SPIN*: solves the overflow problem by allowing more than one index key to be mapped to the same directory entry. (As long as the first few d digits of index keys are the same, they all belong to the same directory entry.)

4. Different index retrieving method:

OKLAHOMA STATE UNIVERSITY

(a) *R_SPIN*: traces the corresponding nodes in the same tree structure, and retrieve the information the nodes contain.

(b) *D_SPIN*: the same as *R_SPIN*.

(c) *T_SPIN*: one directory table corresponding to each level of multidimensional array, and each entry in the directory table contains the file pointer to a certain bucket.

5. Different data file structures:

   (a) *R_SPIN*: records are basic units of data file; many insertions and deletions may cause data file reorganization.

   (b) *D_SPIN*: the same as *R_SPIN*.

   (c) *T_SPIN*: the data file gracefully expands or contracts according the insertion or deletion operations.

OKLAHOMA STATE UNIVERSITY

CHAPTER V

IMPLEMENTATION

An implementation of the two dynamic schemes has been done under UNIX and written in C. Both implementations are based on the two-step transformation method (Chapter IV). The implementation utilizes the assumption that all operations on data occur in main storage. In this chapter, we first present data structures for both schemes, then we give an implementation steps for each dynamic scheme.

Data Structures

Index Dynamic SPIN

The data structure for a single node in the index is shown in Figure 18. The node contains three integer numbers and two structure pointers. The integer left_part contains the K value, which is the initial sequential integer number. The integer right_part contains the V value, which is the computed index number. The integer num_of_child contains the number of children the node has. The two structure pointers are children pointer and parent pointer. A node may have more than one children. Except the top node, each node has one parent node.

Directory Dynamic SPIN

The data structure for directory dynamic SPIN is the same as *R_SPIN*. Both use multidimensional array as their basic data structure.

Index Dynamic SPIN

The *D_SPIN* function implements the basic design idea in Chapter IV. It assumes that all operations on data occurs in main storage. The steps of this program are:

```
struct    {
              int left_part;
              int right_part;
              int num_of_child;
              struct tree *child[ ];
              struct tree *parent;
          }
```

Figure 18: Data structure for D_SPIN

Step IP 1 (Initialization) *Compute the parameters for the formula 2.*

Step IP 2 (Compute index value) *In sparse situation, compute the index value at each level. Put the result in an integer array "rect."*

Step IP 3 (Check root node) *Check whether the root node is NULL; if it it, it needs to be initialized, then go to IP 4. Otherwise, go to Step IP 8.*

Step IP 4 (Root node initialization) *Initialize the root node. Also, initialize the child nodes for root. The left_part of each node is assigned sequential integer number from zero to the maximum number of the nodes. The right_part of each node is assigned zero. Assign the pointers of child nodes to a temporarily pointer array, "temp1." Temp1 points to level zero. Set counter of this level to the number of nodes at this level plus one.*
Set level j*=0.*

Step IP 5 (Go down one level) *Create a temporarily pointer array "temp2", which is a pointer to node at level one. The left_part of each node at level one is the sequential integer number from zero to the maximum number of the nodes. The right_part of each node is initialized to zero. Set counter of this level to the number of nodes at this level plus one.*

Step IP 6 (Connect level j and level j+1) *Allocate memory space for child nodes of temp1. Assign nodes of temp2 to the address of child nodes of temp1; Assign pointers of nodes of temp1 to the pointers of parent nodes of temp2.*

Step IP 7 (Go down next level) *Reinitialize nodes of temp1; Assign pointers of nodes of temp2 to pointers of temp1. Then, go to IP 5, if* $j<=L$ *(L is level number). If* $j>L$, *go to step IP 8.*

Step IP 8 (Retrieve nodes or add new nodes to the trees) *Locate the tree to be retrieved or added by using the subscript at level zero. Set the root of selected tree to subroot. Then, if the execution mode is "r" (retrieve), do Step IP 9; if the execution mode is "a" (add), do Step IP 10.*

Step IP 9 (Retrieve index nodes from the tree) *Starting from the first node at level 1, if the right_part value of the nodes of the subroot's children matches the index subscript at level 1, then, go to next level. We repeat this step until we reach the last level of the tree. At any level of the tree, if the index subscript does not match any nodes at that level, it means that there is no such index value to be retrieved in the index tree, and return -1.*

Step IP 10 (Add new index nodes to the tree) *Starting from the first node at level 1, if we can find the matched node, which means the index subscript value at level one already exists, then we go to next level. If we cannot find the matched node, we create a new path which contains one new node at each level. Set left_part of the new node to the counter value at this level. Update counter value by adding one. Assign right_part of new node the index subscript value. Set number of children of the new node to one. Then, we go to next level, and repeat this search.*

Directory Dynamic SPIN

The *T_SPIN* function implements the basic design idea in Chapter IV. It assumes that all operations on data occurs in main storage. The steps of this program are:

Step DP 1 (Initialization) *Initialize the parameters for formula 2. Set level i=0. Create an integer array "krec" to contain the index values returned.*

Step DP 2 (Compute the index value) *From level 0 to level L, compute the index value at each level by using formula 2. Put the results into "krec." This index value is equal to V value in R_SPIN, which is the result of conversion from the sparse multidimensional subscripts to a one dimensional index value.*

Step DP 3 (Define various parameters) *Define the size of hash table, which will be used in hash function. Also define the value of d, which is an integer number specifying how many digits will be extracted from the hashing result.*

Step DP 4 (Hashing the index value) *Given a hashing function, hash the index value at each level. Store the results into "h_result," which is an integer array. In this program, the hashing function used is:*

$$hash(x) = xmodH_SIZE$$

Step DP 5 (Extract frist d digits from the "h_result") *The hashing result in Step DP 4 is in decimal format. It is transformed into binary format. Then, it is extracted by d digits from its high order end. Put the result into integer array "e_result." Return "e_result."*

CHAPTER VI

PERFORMANCE ANALYSIS FOR SPIN

Testing Program

The testing program experimentally examines the average time complexity of the *D_SPIN* and *T_SPIN* functions. It measures the average running time for insertion and search operations. In order to compare the performances of dynamic SPINs with *R_SPIN*, the average running time of *R_SPIN* is also tested. The steps of the test program are listed below. In the following steps, the test function means *D_SPIN*, *T_SPIN* or *R_SPIN* function.

Step TP 1 (Generate permutations) *Recursively generate distinct multi-dimensional subscript combinations (or permutations), which is used during testing. They are stored in main storage.*

Step TP 2 (Set total test times) *Set a number "N" for the total number of testing cycles. In each testing cycle, a generated permutation is passed to the test function.*

Step TP 3 (Initialize test function) *There are many static parameters involved in the test function, and their computation takes some time. So the test function must be initialized before tests begin.*

Step TP 4 (Measures the overhead) *Since this program tests the average time behavior of test function, the test function is repeated within a testing loop. In this step, a loop is implemented with all the necessary instructions except the functional call of test function.*

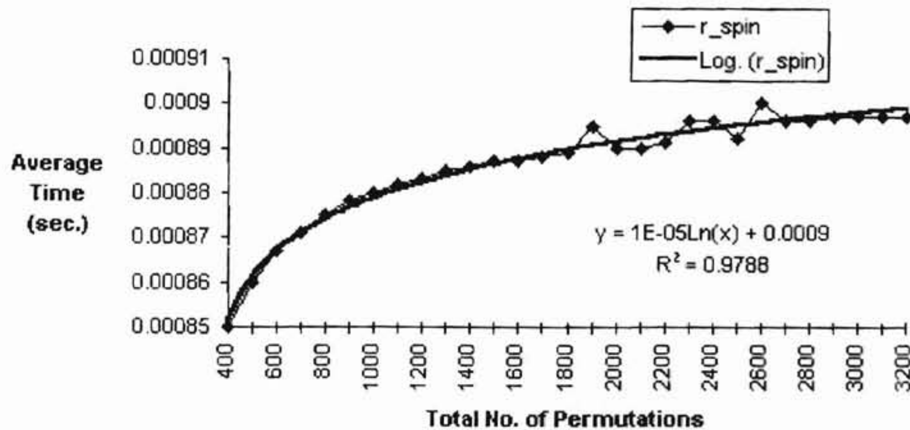


Figure 19: The result of experiment using R_SPIN function

Step TP 5 (Execution loop) *This loop tests all the necessary instructions and execution call to test function. Record the testing time.*

Step TP 6 (Caculate average running time) *Subtract overhead time from the total exaction time, then divide the result by number "N".*

Performance Analysis for R_SPIN

The original *R_SPIN* function is modified so that it operates on the main storage. We assume that the dense dimension size (tree branch size) is eight so that given a total number of permutations (3200 in this example), the probability of overflow situation is relatively small. All graphs in this chapter are created by Microsoft Excel. Figure 6.1 shows the experimental results of *R_SPIN* with a permutation definition of five dimensions and dimensional size of eight in each dimension.

In Figure 19, "Total No. of Permutations" refers to the different total permutation numbers used in this experiment, and "Average Time" means the average insertion time.

Figure 19 shows that the average insertion time of *R_SPIN* function is logarithmic. The trendline function is logarithmic function. The R-square of the trendline is 0.9788 (close to 1), which means the simulation of trendline is good.

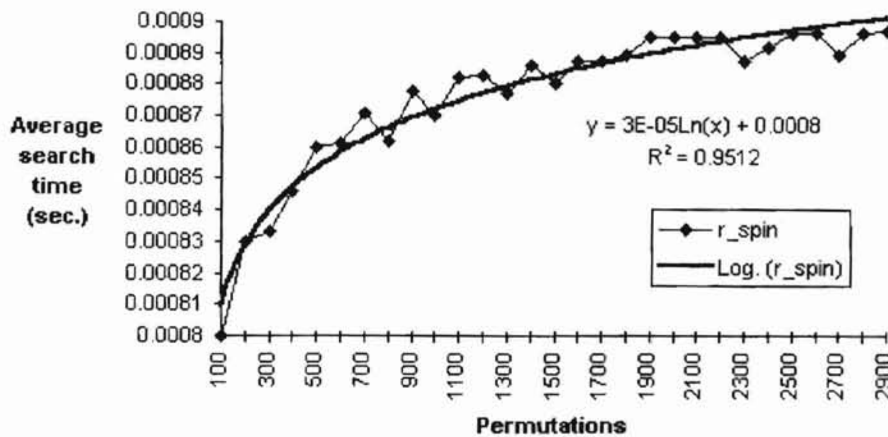


Figure 20: The result of experiment using R_SPIN function

This average insertion time does not include the initialization time for index trees. It only counts the insertion time on initialized index trees. The theoretical proof shows that average time complexity for insertion or search on this kind of tree structures (incomplete tree) has logarithmic behavior[12].

The search operation in R_SPIN

The search experiments in this example use some of the permutations that were inserted in earlier. As a result, all search operations in the experiments are successful. Figure 20 shows the results of experiment with dimensional size eight. The experiment indicates that the average search time of the *R_SPIN* function has logarithmic time complexity.

Performance Analysis for D_SPIN

We assume that the operations occur in main storage, the binary tree is the initial index data structure for *D_SPIN*, the level number for index tree is four (dimension number is five). Figure 21 shows the experimental results of *D_SPIN*.

Figure 21 shows the average insertion time of *D_SPIN* is a logarithmic time complexity. The function of trendline is logarithmic time complexity. The value of R-square is close to one (0.9553), which means a good simulation of the trendline.

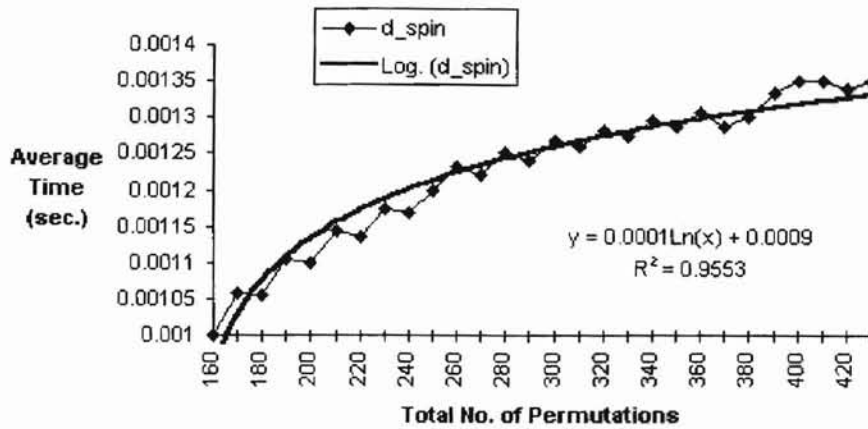


Figure 21: The result of experiment using D_SPIN function

In view of algorithm design, there are several factors contributed to this logarithmic average insertion time.

1. Initialization time is not included in the test result. The initialization time means time for creating initial tree structures. It usually cost much more than that for operations (insertion or search) on initialized trees. In D_SPIN, besides the initialization time, there is also some time needed to create new nodes when overflow happens. The total number of those new nodes depends on how many nodes are created and initialized at the beginning and how often the overflow happens. Generally, the more new nodes are initialized at the beginning, the less new nodes are needed to be created during execution.

2. In practical applications, since the probability of overflow can be predicted roughly, the number of initialized nodes should at least count 50% of total permutations so that less time will be spent on creating new nodes and general performance of *D_SPIN* is acceptable. In this example, it counts 37%. If there is no overflow then it is reasonable for *D_SPIN* having a logarithmic execution behavior, because the *D_SPIN* executes on the same index tree structure as *R_SPIN*, and *R_SPIN* has been proved to have a logarithmic average time com-

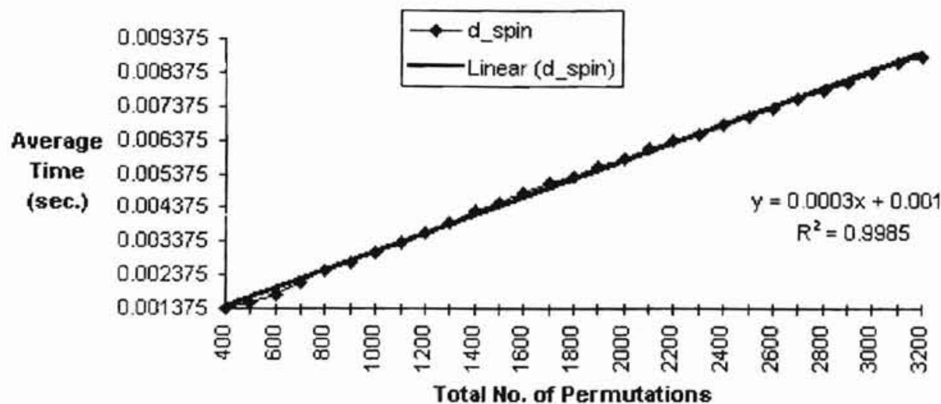


Figure 22: An experiment using D_SPIN function

plexity. If the time for creating new nodes does not count very much in total execution time, it can be expected that average insertion time of *D_SPIN* has logarithmic behavior.

3. In the implementation of *D_SPIN*, when an overflow at the same level happens, not only a new node at that level is created, but also a path of child nodes is created. So, it effectively saves the time for creating new nodes when overflow happens on the same path repeatedly.

4. In some cases, if the time needed for creating new nodes counts a much bigger percentage in total execution time, then it can be expected that the average insertion time of *D_SPIN* is linear. Figure 22 confirms this expectation.

   In Figure 22, the number of initialized nodes only counts 3.8% of total permutations. We see that trendline is a linear function.

The search operation in D_SPIN

The search experiments in this example use some of the permutations that were inserted in earlier. As a result, all search operations in the experiments are successful. Figure 23 shows the results of experiment with dimensional size two. The experiment

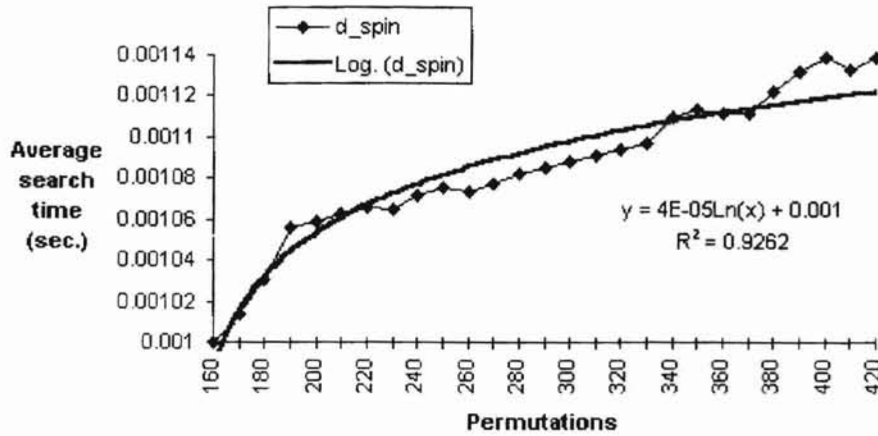


Figure 23: A search experiment using D_SPIN function

indicates that the average search time of the *D_SPIN* function has logarithmic time complexity.

Compare D_SPIN with R_SPIN

If we compare the trendline function of *D_SPIN* with that of *R_SPIN* for insertion operation, we find the fixed parts of both functions are the same; but for variable parts (value of slope), the *D_SPIN*'s is ten times larger than the *R_SPIN*'s. In other words, R_SPIN is ten times faster than *D_SPIN*.

$$y = 0.0001Ln(x) + 0.0009.....d_spin \tag{3}$$

$$y = 1E - 05Ln(x) + 0.0009.....r_spin \tag{4}$$

Considering the static property of *R_SPIN* and dynamic property of *D_SPIN*, it is reasonable that execution of *R_SPIN* is several orders of magnitude faster than execution of *D_SPIN*.

Performance Analysis for T_SPIN

As stated in Chapter IV, the *T_SPIN* function is actually a transformation function. It transforms a sparse multidimensional array into a dense multidimensional array, which is further transformed into d binary digits. The data structure the

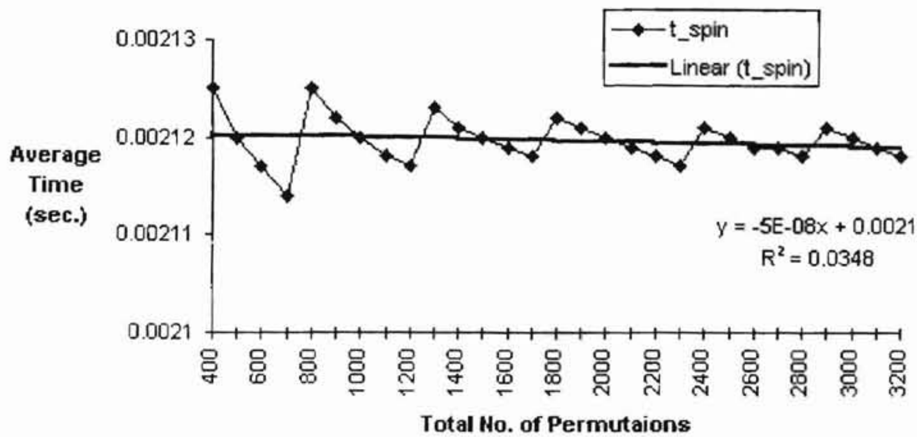


Figure 24: The result of experiment using T_SPIN function

T_SPIN function uses is multidimensional array, which is similar to having only one path in *D_SPIN*'s tree structures. So, given a certain level of multidimensional array. a constant average running time can be expected. This expectation is confirmed by experiments. Figure 24 shows the experiment results.

In Figure 24, the slope of trendline function is very small (-5E-08) and can be considered close to zero. So, the average execution time of *T_SPIN* can be considered to be constant. In this case, it is 0.0071 seconds. We also see from Figure 24 that larger the number of input data is, more the average running time is close to 0.0021. One reason for this constant time complexity is that the *T_SPIN* does not keep directory table which is necessary for data record retrieving and search. The output of *T_SPIN* is the index value for the directory table. The management of directory table is left for file system of database.

The other way to measure the performance of *T_SPIN* is to analyze the distribution of its output. The output of *T_SPIN* is the d binary digits, which is required to evenly distributed in the directory table, so that the load factor of each bucket is closed to each other. One important factor to influence the values of d binary digits is the hashing function used in *T_SPIN*. In this test program, *T_SPIN* uses a typical hashing function:

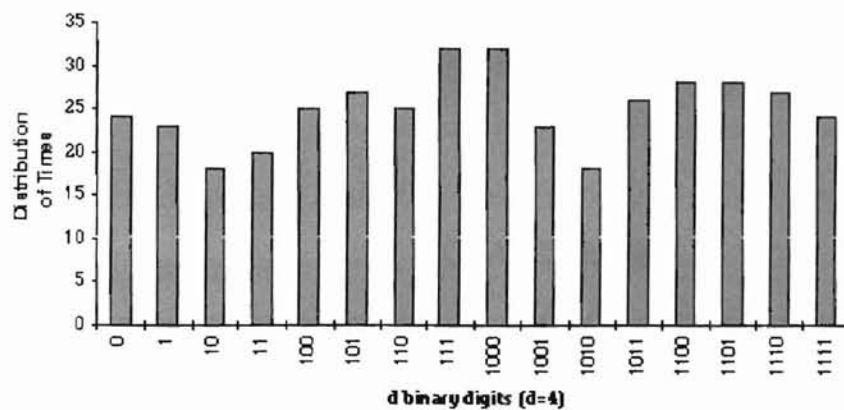


Figure 25: The result of experiment using distribution of output in T_SPIN

$$hash(x) = x mod H_SIZE$$

Figure 25 shows the experimental results of *T_SPIN*, using d=4 at level 4.

From Figure 25, we see that the distribution of output is good. No directory entry has too many or too few index numbers.

CHAPTER VII

SUMMARY, CONCLUSION, AND SUGGESTED FUTURE WORK

Index dynamic SPIN (*D_SPIN*) and directory dynamic SPIN (*T_SPIN*) are two dynamic indexing techniques that do not require complete file reorganization as a result of overflow. They can be very useful for applications where overflow occurs often, and the frequency of overflow cannot be predicted accurately. They also eliminate the sparse situation in index files so storage space is saved when the index data file is large. The design of both dynamic SPINs follows the two-step transformation method, which is presented in chapter IV.

Conclusion

D_SPIN employ an index tree structure in the index file. It keeps all necessary information in nodes of the tree. The advantage of this method is that it keeps the layered data structure. The disadvantage of this method is that it must create new nodes and a new path to overcome overflow; thereby, reducing the efficiency of operations when compared to the static SPINs. *T_SPIN* employs a hash function in its transformation processes. The output of *T_SPIN* is d binary digits, which will be further used by file system to retrieve specific bucket. The advantage of *T_SPIN* is that it introduces bucket concept, and it keeps data file dynamic since buckets in the data file split on overflow. The disadvantage of *T_SPIN* is that it sacrifices execution speed since it needs to hashing and extract the outputs instead of directly retrieving outputs as does *R_SPIN*.

The empirical results presented above demonstrate that a trade-off exists between dynamic SPINs and static SPINs. If time is a major factor, static SPINs show

better performance since static methods are almost ten times faster than dynamic SPINs in the test example. However, if eliminating overflow and saving storage space is a premium, then dynamic SPINs show better performance because there is neither overflow nor sparse data storages in dynamic SPINs.

Suggested Future Work

The work in this thesis is limited to the indexing technique which is only one link in whole database design chain. It is necessary to measure the performance of dynamic SPINs in practical database design and application. For example, after getting results from *T_SPIN*, one must design and implement buckets, file pointers, directory tables, ..., etc.. so that the test constitutes a complete picture of a database design. This part of work is left to future study.

A SELECTED BIBLIOGRAPHY

1. Guttman, A., *R-trees: a dynamic index structure for spatial searching*, Proceedings of ACM SIGMOD (Special InterestGroup on the Management of Data), June, 1984, pp.47-57

2. Bayer, R. and M. Schkolnick, *Concurrency of operations on B-trees*, Acta Informatica 9, 1977, pp.1-21

3. Coburn, Ty K. *C SPIN toolkit*, Oklahoma City, OK: ca.1991.

4. Coburn, Ty K. *An introduction to SPIN hashing: an approach to managing multidimensional data spaces.* Tinker AFB, OK: Unpublished technical report, ca. 1991.

5. Coburn, Ty K. *An introduction to the S_SPIN hash function: making more out of the multidimensional array.* IEEE (Institute of Electrical and Electric Engineers) NAECON(National Avionics Engineering Conference) 1994.

6. Fan, Z.M., Y. Zhang, R.A. DiVall and G.E. Hedrick *Overflow analysis in SPIN.* Computer Science Dept., OSU: Unpulished technical report, OSU-CS-TR-96-01, 1996.

7. Fagin, R., J. Nievergelt, N. Pippenger, and H.R. Strong *Extensible hashing - A fast access method for dynamic files.* ACM Transactions On Database Systems, Vol 4. pp. 315-344. Sept. 1979.

8. Larson, P. A. *Dynamic hashing.* BIT 18, pp. 184-201, 1978.

9. Flajolet, P. *On the performance evaluation of extendible hashing and trie searching.* ACTA Informatica, Vol 20, pp. 345-369, 1983.

10. Korth, H. F. and A. Silberschatz *Database System Concepts.* New York: McGraw-Hill, Inc., c1991.

11. Zhang, Y., R.A. Divall, M.Z. Fan and G.E. Hedrick *An Experimental Analysis of a New Multi-Dimentional Storage And Retrieval Method.* Computer Science Dept., OSU: Unpulished technical report, OSU-CS-TR-95-04, 1995.

12. Divall. R.A., Y. Zhang, M.Z. Fan and G.E. Hedrick. *A Theoretical Analysis of a New Multidimentional Storage and Retrieval Method.* Computer Science Dept., OSU: Unpulished technical report (in preparation), 1995.

13. Harbron, T. R. *File systems: structures and algorithms.* Englewood Cliffs NJ: Prentice Hall, Inc. c1987.

14. Aho, A.V., J.E. Hopcroft, and J.D. Ullman *Data Structures and Algorithms.* Reading, MA: Addison-Wesley, 1983.

15. Aoe, J., Y. Yamamoto, and R. Shimada, *A Practical method for reducing sparse matrices with invariant entries*, International Journal of Computer Mathematics, Vol. 12, pp. 97-111, Nov. 1982.

16. Boyer, R.S. and J.S. Moore, *A fast string searching algorithm*, Communication of the ACM, Vol.20, No.10, pp. 762-772, Oct. 1977.

17. Buehrer, D.J. and Y.W. Fan, *SL-trees: An indexing structure for object-oriented data-bases.* The Journal of Systems and Software, Vol.32, No.3, pp. 237-249, Mar. 1996.

18. Chang, Y. and Lee C., *Climbing hashing for extensible hashing*, Information Science, Vol.86, No.3, pp. 77-99, Sept. 1995.

19. Cormack, G.V., R.N.S. Horspool and M. Kaiserswerth, *Practical perfect hashing*, The Computer Journal, Vol.28, pp. 54-58, Jan. 1985.

20. Jacobs, D.W., *The space requirements of indexing under perspective projections.* IEEE Transactions on Information Theory, Vol.18, pp. 330-333, Mar. 1996.

21. Jaeschke, G., *Reciprocal hashing: A method for generating minimal perfect hashing functions*, Communication of the ACM, Vol.24, pp. 829-833, Dec. 1981.

22. Jonge, W.D., A.S. Tenenbaum, and R.D. Reit, *Two access methods using compact binary trees*, IEEE Trans. Software Engineering, Vol. SE-13, pp. 799-810, Jul. 1987.

23. Knuth, D.E., *The Art of Computer Programming*, Vol. III: *Sorting and Searching.* Reading, MA: Addison-Wesley, 1977.

24. Knuth, D.E., J.H. Morris, and V.R. Pratt, *Fast pattern matching in strings*, SIAM Journal of Computer, Vol. 6, No. 2, pp. 323-349, Jun. 1977.

25. Kumar, V. and J. Mullins, *An integrated data structure with multiple-access paths for database-systems and its performance.* Data and Knowledge Engineering, Vol.16, No.1, pp. 51-72, Jul. 1995.

26. Maly, R., *Compressed Trees*, Communication of the ACM, Vol. 19, No. 7, pp. 409-415, Jul. 1976.

27. Orenstein, J.A., *Multidimensional tries used for associative searching*, Information Processing Letters, Vol. 14, No. 4, pp. 150-157, Jun. 1982.

28. Sheil, B.A., *Median split trees: A fast lookup technique for frequently occurring keys*, Communication of the ACM, Vol. 21, pp. 947-959, Nov. 1978.

29. Standish, T.A., *Data Structure Techniques*. Reading, MA: Addison-Wesley, 1980.

30. Tarjan, R.E. and Yao A.C., *Storing a sparse table*, Commun. ACM, Vol. 22, pp. 606-611, Nov. 1979.

31. Wirth, N., *Algorithms and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

APPENDIX A

Glossary

Any computer program whose objective is the solution of a practical problem.

application environment

An environment imputed to the SPIN package by the application program.

dense[tree])

A tree whose nodes all contain data.

dynamic SPIN

Any method that is contained in the SPIN package that allows the structure of the data to change dynamically.

key distribution

The way the keys of data objects are distributed throughout the data space. Frequently expressed as a probability distribution function.

level [in SPIN]

The part of a storage tree that corresponds to a given index in a multidimensional array.

load factor

A number, λ, whose value is between 0 and 1, and whose value indicates how full (the load) the storage tree(or table) is. An empty tree (table) has a load factor of 0 (λ=0); a full tree (table) has a load factor of 1.

multidimensional array

Any array with more than one dimension.

overflow

In SPIN, an attempt to insert data into a tree with a load factor of 1.

radix[tree]

A tree that has a fixed number of [possibly empty] branches from each node. A radix n tree has n branches at each node.

sparse [data]

Multidimensional data in which the number of zero or empty storage locations greatly exceeds the number of nonzero, nonempty storage locations.

static SPIN

Any method that is used within the SPIN package and whose storage structure remains fixed after initial allocation.

SPIN

Single Point Index Network. A data structure for data organization and retrieval.

SPIN mapping

Any mapping of keys, index values, or data values within one of the SPIN methods.

data file

A file is a collection of records.

bucket(or page)

A bucket (or page) corresponds to one or several physical sectors of a secondary storage device such as a disk. Let the capacity of a bucket be b records.

space utilization

*Space utilization is the ratio between n and m*b, where n is the number of records in the file, m is the number of pages used, and b is the capacity of the page.*

VITA

ZILI FAN

Candidate for the Degree of

Master of Science

Thesis: DYNAMIC SPIN SCHEMES

Major Field: Computer Science

Biographical:

Education: Graduated from Hangzhou Second High School, Hangzhou, China in July, 1987; received Bachelor of Science degree in Management Science from Hangzhou University, Hangzhou, China in July, 1991; received Master of Science degree in Economics from Oklahoma State University, Stillwater, Oklahoma in December, 1994.
Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 1996.

Experience: Employed by Oklahoma State University, Department of Computer Science as a graduate research assitant, 1995 to 1996