96-GP ✓

Performance Evaluation of G-tree and Its Application in Fuzzy Databases*

Chengwen Liu
DePaul University, Chicago, Illinois
liu@cs.depaul.edu

Aris Ouksel, Prasad Sistla,
Jing Wu, Clement Yu
University of Illinois at Chicago
yu@eecs.uic.edu

Naphtali Rishe
Florida International University

Abstract

G-tree is a data structure designed to provide multi-dimensional access in databases. It has the self-balancing property of B^+-tree. In this paper, performance evaluation of G-tree is provided for various data distributions. For point queries, the experiments show that its retrieval and update performance is similar to that of B^+-trees independently of the data distribution. For range queries, the performance varies significantly with the data distributions. While the performance is good for the 2-dimensional case, it deteriorates as the number of dimensions increases. This empirical evidence is confirmed by an analytical proof, which also yields a simple way of computing the expected number of data pages accessed. This analytical result which shows that the number of data pages accessed by a range query increases exponentially with the number of dimensions applies to many multi-dimensional schemes. We also apply the G-tree to fuzzy databases and show empirically that it has good performance for imprecise queries on relatively imprecise data. But it is less efficient for precise queries on relatively precise data.

1 Introduction

G-tree [7] is a structure designed to handle multi-attribute queries. It combines features of both grid files [5, 6, 10, 14] and B^+-tree. The multi-dimensional space is divided into a grid of variable size partitions, as was done in [5, 14]. Each partition is identified by a unique number. This number is a bit string which retraces the binary splitting history from the original data space to the specific partition. A similar numbering scheme was used in [2, 11, 13, 15]. The partition numbers are stored in a B^+-tree according to a lexicographic order. In this respect, it is similar to [5, 11]. The partitions are associated with disk pages or buckets that are linked to the B^+-tree. Each bucket stores the data points (records) belonging to the corresponding partition. Since B^+-trees are balanced, retrieving a bucket with a given partition number is efficient.

The use of B^+-tree is very widespread in many commercial database implementations. Therefore, any efficient multidimensional structure which relies on this structure can be easily incorporated into these implementations. This is the case of the G-tree. It is similar to B^+-tree and thus can be implemented with ease. In this paper, we evaluate the performance of G-tree under various data distributions and study its application in fuzzy databases.

The remaining of the paper is organized as follows. In section 2, we describe G-tree and the algorithms for searching. In section 3, we present experimental results for different data distributions. In section 4, we present the theoretical performance result on d-dimensional searching. In section 5, the application of G-tree to fuzzy databases and temporal databases is discussed. Experimental results for fuzzy databases are provided and analyzed. The conclusion is given in section 6.

2 G-tree(Grid tree) Description

2.1 G-tree Partition Method

Initially, the entire data space consists of a single partition. When a bucket overflows due to an insertion, the corresponding partition is split into two equal sub-partitions. The bucket associated with the original partition is assigned to one of the sub-partitions and a new bucket is assigned to the other. If all data points fall within one of the sub-partitions, which will cause the corresponding bucket to overflow, the sub-partition will be split. This procedure recurses until there is no overflow. The partitioning is performed by repeatedly and cyclically splitting the partition in half along the d dimensions. The partition number of a partition is a bit string tracing the sequence of splits that led to the partition from the original space.

In the following, we use a simple example to explain the basic idea of G-tree. Each node of G-tree contains up to m entries. An entry of an internal node consists of

*This research was supported in part by NASA (under grant NAGW-4080), ARO (under BMDO grant DAAH04-0024), NSF (under grant NSF-IRI-95 09253) and LAS of DePaul University.

CIKM 96, Rockville MD USA
© 1996 ACM 0-89791-873-8/96/11 ..$3.50

a partition number and a pointer to a child index page. An entry of a leaf node consists of a partition number and a pointer to the corresponding data page. Similar to B^+-tree, each node has a pointer to its right sibling.

Example 1

Fig 1.(a) shows a possible partitioning of a two-dimensional space. The partition numbers are maintained in the G-tree as shown in Fig 1.(c). Consider the insertion of data points into partition "1" until it overflows. Since partition "1" was obtained by splitting the original space along the first dimension, it will now be split along the second dimension as shown in Fig 1.(b) according to the cyclic splitting order. Specifically, it is split into partitions "10" and "11", which are assigned one data page each. Partition number "1" is now replaced by two numbers in the index page, causing an overflow. From now on, overflow is handled as was known in B^+-trees. Fig 1.(d) shows the state of the G-tree after this new partition.

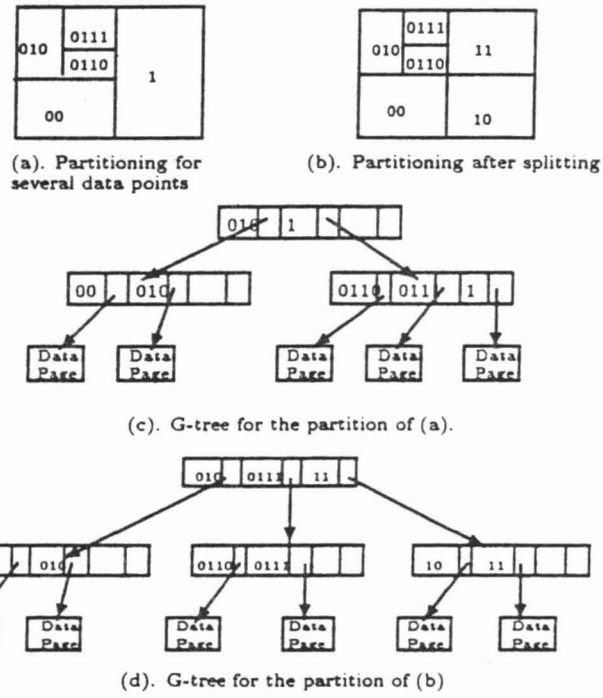

Figure 1: Partitions and G-trees for Example 1

The main features of the G-tree are given as follows:

1. The data space is divided into non-overlapping partitions of variable size.
2. Each partition is assigned a unique partition number (binary string).
3. The partition numbers are stored in a B^+-tree based on their natural ordering.
4. Empty partitions are not stored in the tree to save space.
5. The partitions in a G-tree do not overlap and are totally ordered by the $<$ relation.

Let $P_1(b_1$ bits long) and $P_2(b_2$ bits long) be two partitions, and $b = min(b_1, b_2)$. The following definitions are based on comparing the b Most Significant Bits(MSB) of P_1 and P_2.

Definition 1 (Greater) : $P_1 > P_2$ if $MSB(P_1, b) > MSB(P_2, b)$.

Definition 2 (Ancestor): $P_1 \supset P_2$ if $MSB(P_1, b) = MSB(P_2, b)$, and $b_1 < b_2$.

Definition 3: $P_1 \supseteq P_2$ if $P_1 \supset P_2$ or $P_1 = P_2$.

Definition 4: The parent partition of P, Parent(P), is determined by removing the last bit from P.

Definition 5: The complement of partition P, compl(P), is determined by inverting its last bit.

Note: If a child's partition P_{child} exists, then none of its parents' partition exists.

2.2 Searching using G-tree

In this subsection, we present the G-tree searching algorithms for point queries and range queries. Insertion and deletion algorithms are not difficult to construct [7], but are not presented here.

1. **Exact Match Queries(Point Queries)**
   (a) Compute the initial partition number P_{in} of query point Q (assuming that Q will be in a partition with the dimensions equal to those of the smallest partition created so far).
   (b) If Root is null, terminate. Otherwise, let X = Root.
   (c) While X is an internal node, find the first partition P such that $P > P_{in}$ *or* $P \supseteq P_{in}$, and let X = child(P).
   (d) Find the partition P such that $P \supseteq P_{in}$(P is unique if it exists), then each point in it must be examined and checked individually. Output the query results. If such P does not exist, search fails.

   Note: In the above searching process, (c) and (d) can use binary searching method to find P.

2. **Range Queries**
   (a) Identify the query region's smallest and largest query points (the opposite corners of the query region), and compute the partition numbers P_{lo} and P_{hi}, whose partitions contain the smallest and the largest query points respectively.
   (b) For each partition number, apply **Exact Search Method** to get the actual partition P_{act_lo} for P_{lo} and P_{act_hi} for P_{hi} ($P_{act\_lo} < P_{act\_hi}$).

(c) All partitions lying in this range(P_{act_lo}, P_{act_hi}) are tested to determine whether they are fully contained in the query region, overlap the query region, or are outside the query region. For each partition number within P_{act_lo} and P_{act_hi}, it is transformed to its corresponding region, and then compared with the query region.

i. Fully contained, then all points in it satisfy the query.

ii. Overlap, then each point in it must be checked individually.

iii. Outside, the partition will not be considered any further.

Then output query results.

3 Performance Evaluation of G-tree

Performance studies of other structures were given in [8, 17]. In this section, we provide a thorough evaluation of the performance of the G-tree structure for various data distributions.

3.1 Data Distributions

In order to be able to compare our results with earlier ones, the same data distributions as those of [8] are used in our experiments. The data space was normalized to be $[0, 1]^2$ and there are 10^5 data points without duplicates. In the following, $U(a, b)$ denotes the uniform distribution between a and b and $N(m, v)$ denotes the normal distribution with mean m and variance v.

(Dist1) Diagonal: The points follow a uniform distribution on the main diagonal.

(Dist2) Sine Distribution: The points follow a sine curve. The points are uniformly distributed on the x-axis and the y coordinates follow $N(sin(2\pi x), 0.1)$.

(Dist3) x-Parallel: The x coordinates of the points are uniformly distributed and y follow $N(0.5, 0.01)$.

(Dist4) Clustered Points: The x coordinates of the points follow $N(z_x, 0, 05)$ and y follow $N(z_y, 0.05)$, where (z_x, z_y) are the coordinates of the cluster centers. Here, we arbitrarily choose ten cluster centers which are located at the lower left triangle of the data space, with six of the cluster centers located around the lower left corner of the data space. Each group of clustered points contains 10000 points.

(Dist5) Uniform Distribution: The points are uniformly distributed over $[0, 1]^2$.

3.2 Experiments

For each of the five data distributions, experiments were conducted to evaluate the storage utilization, the performance of insertion, deletion and queries (including range queries and exact match queries). For range queries, we considered the following five types:

Q1: range queries in which the queries are squares with area 0.001 in which the centers of the squares follow a uniform distribution over $[0, 1]^2$.

Q2: range queries in which the queries are squares with area 0.01 in which the centers of the squares follow a uniform distribution over $[0, 1]^2$.

Q3: range queries in which the queries are squares with area 0.1 in which the centers of the squares follow a uniform distribution over $[0, 1]^2$.

Q4: partial match queries in which the two specified x coordinates in each query are uniformly distributed over $[0, 1]^2$ and the y coordinates are unspecified.

Q5: partial match queries in which the two specified y coordinates in each query are uniformly distributed over $[0, 1]^2$ and the x coordinates are unspecified.

For each query type, we execute 20 randomly generated queries and compute the average number of disk accesses and the average time. All the experiments were conducted on a SUN SPARC-2 workstation with 48MB memory and 4GB disks. The page size was set to 1024 bytes(1K).

3.3 Analysis of Experimental Results

In order to describe our experimental results, we first define the following parameters:

m: the maximum number of records in a page $m = \frac{page\ size - page\ header\ size}{tuple\ size}$, where page header size is 16 bytes in our implementation.

Page utilization: $\frac{number\ of\ entries\ in\ bucket}{m}$.

3.3.1 Experiment 1: storage utilization

The *first* experiment was performed to investigate the page utilization of the G-tree structure for the five data distributions and various tuple sizes. The size of each attribute was set to 4 bytes. As the number of attributes in each tuple (data point) changes from 2, 4, 8, to 15, the value of m varies from 126, 63, 31, to 16 accordingly. The results are given in Table 1. The average utilizations for non-uniform distributions (**Dist2**, **Dist3** and **Dist4**) are around 69%, and they are more or less independent of the value of m. The average utilizations for the uniform distributions(**Dist1**, **Dist5**) are higher than 69% and increase to 77% as m increases to 126. When a data page overflows, uniform distributions permit better storage utilizations than non-uniform distributions because of the G-tree's partitioning method, which splits a partition into halves.

A similar experiment was performed in [7], where 50,000 data points (2 dimensions) were used and m is set to three values: 25, 50 and 100. Our results agree with those of [7].

m	Dist1	Dist2	Dist3	Dist4	Dist5
126	77.56%	68.51%	68.57%	69.15%	77.49%
63	76.33%	69.17%	68.54%	69.32%	76.25%
31	73.16%	69.27%	69.49%	69.33%	73.23%
16	70.73%	69.38%	69.11%	69.49%	70.57%

Table 1: Data page utilization for various values of m

	Dist1	Dist2	Dist3	Dist4	Dist5
ipu	56.4%	56.5%	56.5%	78.2%	53.1%
io	3.93	3.93	3.92	3.92	3.93
t(ms)	3.15	3.25	3.21	3.22	3.17

ipu - index page utilization; t - average time per insertion
io – average # of disk accesses

Table 2: Performance for insertion of 10^5 data points

3.3.2 Experiment 2: Insertions

The *second* experiment tests the performance of G-tree for insertions with various m values. 10^5 points are randomly inserted into the database. Tables 2 shows the results for $m = 126$.

From Table 2, we can easily see that the index page utilizations are around 55% with the exception of **Dist4**. For **Dist4**, points are concentrated at a few locations, many partitions are empty and are not stored in the G-tree. As a result, the index utilization is much higher (78.18%). The average number of disk accesses is approximately 3.92. For insertions, the performance of the G-tree is more or less independent of the data distributions. In this experiment, the height of the G-tree is 2. Thus, inserting a point at *height* $= 2$ requires at least 2 disk accesses for reading the index pages, 1 disk access for reading the data page and 1 disk access for writing the data into the page, requiring a total of 4 disk accesses. When the G-tree is of height 1, the number of disk accesses is 3. Since there are a lot more insertions when *height* $= 2$ than when *height* $= 1$, the average is close to 4. This implies that the overhead caused by splitting data and index pages is negligible. We also observed that the elapsed time is only about 3.2ms which is rather low compared with the number of disk accesses. This is because the operating system buffering is in effect and the buffers were not cleared between queries (i.e., the cache was hot during the experiment).

When $m = 16$, the index page utilizations are higher with the exception of **Dist 4**. But the changes are not drastic. The average number of disk accesses is 4.17. In this case, the height of G-tree remains 2. However, as the number of records per data page decreases from 126 to 16, the chance for a data page and its ancestor index pages to split per inserted record increases, resulting in more page accesses. However, the increase is only 6.4%. The average elapsed time per insertion also increases from 3.2ms to 5.35ms, since as m decreases from 126 to 16, there are many more data pages to be written back to secondary memory.

3.3.3 Experiment 3: Range Queries

This experiment tests the performance of range queries. The database contains 10^5 points and m varies from 16 to 126. Because of space limitation, we only show the result for $m = 126$ in Table 3. From the experimental results, we make the following observations:

1. For range queries, as the size of the region covered by the query increases, the number of disk accesses increases, but the increase is less than linear. For example, when the region size increases 10 times from Q1 to Q2 or from Q2 to Q3, the number of disk accesses increases from 3 to 7 times only. This can be explained by the following analysis.

   Let the partition containing the query region's smallest point be P_{lo} and the partition containing the query region's largest point be P_{hi}. Assume the index page pointing to the data page of P_{lo} be I_{lo} and that of P_{hi} be I_{hi}. The processing of a range query involves searching of the index pages I_{lo} and I_{hi}. If the height of the G-tree is h, then searching of the two index pages takes $2*h$ disk accesses. Each index page from I_{lo} to I_{hi} is then examined to determine whether it contains a pointer to a data page needed by the query. We assume the number of index pages from I_{lo} to I_{hi} inclusive is T. Then, the total number of index page accesses is $2*h+T-2$. As a result, the average number of disk accesses for a query covering r percent of the data space can be estimated by

   $$2h + T - 2 + \frac{nr}{\mu}$$

   where n is the total number of data pages, μ the average data page utilization, and $\frac{nr}{\mu}$ the average number of pages containing the data. While the number of data pages ($\frac{nr}{\mu}$ in the above formula) is directly proportional to the region size of the query, the index pages increases sub-linearly with respect to the region size.

2. For partial queries, there is very little difference as a result of changing the unspecified attribute, except for **Dist 3**. In the latter case, data is concentrated at $y = 0.5$, and thus when y is specified and its range contains 0.5, the answer covers a lot more data points than when x is the only specified attribute. As a result, the number of disk accesses is higher.

3. The effect of data distributions on performance is mild relative to that due to the query types.

4. In [8], performance comparisons were provided for four spatial indexing methods including the 2-level grid file [6], BANG file [5], the hB-tree [9] and the Buddy Hash Tree (BHT) [8]. Among the competitors, BHT was found to be the winner. It exhibits an at least 20% better average performance than the other methods. Here, in Table 4, we provide a comparison of G-tree to BHT for **Dist1** and **Dist4**. For the other data distributions, only

	Dist1	Dist2	Dist3	Dist4	Dist5
Q1	6.8	8.65	7.7	8.35	8.5
Q2	18.45	26.35	18.7	23.0	22.55
Q3	137.9	136.1	226.3	165.5	133.5
Q4	391.71	470.85	468.0	466.65	426.15
Q5	392.8	462.4	675.25	466.95	427.95

Table 3: Average number of IOs for queries on a G-tree of 10^5 points (m=126)

	Dist1		Dist4	
	BHT	G-tree	BHT	G-tree
Q1	1	6.8	4	8
Q2	34	18.5	4	23
Q3	297	137.9	242	165

Table 4: Comparison of G-tree with BHT (# of IOs)

normalized performance data were provided in [8], thus it is impossible to compare. It is obvious from Table 4 that G-tree performs better than BHT for queries retrieving substantial amount of data while its performance for queries retrieving very small amount of data is not as good as those of the other methods [8].

3.3.4 Experiment 4: Deletion

In this experiment, 10^4 instructions for deletions are executed for the database having 10^5 points ($m = 126$). Since the height of G-tree is 2, we need 3 disk accesses for reading (2 index pages + 1 data page) and 1 disk access for writing the updated data page back if the data point is actually deleted. Some instructions may not result in deletions, because the points to be deleted are absent. But some other deletions cause index pages to be updated. For **Dist4**, only 10^3 data points are actually deleted, and thus the time taken by writing back the data pages are reduced. Its average number of disk accesses is 3.1 and average elapsed time is 3.5ms. For the other data distributions, the number of points actually deleted ranges from 9253 to 10000. The number of IOs is about 4 and the execution time is about 4.2ms.

3.3.5 Experiment 5: Exact Match Queries

In this experiment, we search for 10^4 data points in the database with $m = 126$. As expected, the number of disk accesses is the height of the G-tree plus 1. The average time is around 3.0 ms for each point query. The variation is imperceptible across the data distributions.

3.3.6 Experiment 6: Three Dimensional Cases

In this experiment, the number of data points is increased to 200,000 and $m = 12$. Insertions, deletions, point queries, range queries and partial queries are tested for both the two-dimensional and three-dimensional cases. The partial and the range queries for the 3-dimensional case are generated such that each of them retrieves the same proportion of data as their counterpart in the 2-dimensional case.

	number of IOs		time (sec.)		index pages		useless idx pages	
dim.	2	3	2	3	2	3	2	3
Q1	51	111	0.6	1.5	12	47	9	41
Q2	314	457	1.65	2.0	34	73	26	58
Q3	2633	3218	6.1	7.25	116	190	72	117

Table 5: Performance of range queries ($2 * 10^5$ points, m=12)

Our experimental results show that for insertion, deletion and point queries there is little difference for 2 and 3 dimensions. Data page utilizations for both cases are about the same (70%). Index page utilizations are 76% and 74.9% respectively. The average number of disk accesses per insertion is 4.8 and the execution time is about 10.5ms. For point queries, the average number of disk accesses is 4 and the execution time is approximately 5.5 ms. The average number of disk accesses for deleting a record is around 5.0 and the time is around 10 ms. Since the height of the index tree is 3, searching takes 4 disk accesses, deleting takes around 5 disk accesses as writing the data page takes an additional access.

Table 5 shows the results for the 3 types of range queries. It is obvious that the average number of IOs increases drastically when the number of dimensions increases from 2 to 3, especially for queries retrieving a small percentage of data. For Q1 which retrieves only 0.1% of the data, the number of disk accesses increases by more than a factor of 2.

The table also shows that the number of index pages traversed in 3 dimensions is larger than that for 2 dimensions. This is due to the fact that the low and the high data points in 2 dimensions are separated by fewer partitions than the corresponding points in 3 dimensions. The numbers of useless index pages traversed from the low data points to the high data points are also exhibited, where an index page traversed from the low data point to the high data point is useless if it does not point to a data page required by the query.

4 Performance Analysis of Range Queries

One of our observation is that the number of pages retrieved per range query increases exponentially with the number of dimensions in multidimensional structures. In this section, we develop an analytical model to evaluate the retrieval cost of range queries and prove the validity of our observation.

Let **N** be the total number of points in the data space, **b** be the maximum number of data points that can be placed in a page, and μ be the utilization of data pages. Then the average number of data pages to hold **N** data points is: $\lceil \frac{N}{\mu b} \rceil$.

Let x denote the fraction of accessed pages to answer a range query. It can also be viewed as the fraction of the data space covered by the pages containing the answer. We first analyze the one-dimensional case. For simplicity, the data space, which is a bounded segment

of the real line, is normalized to length 1. Assuming a uniform distribution, each page can be viewed as an interval of length $y = \frac{1}{\lceil \frac{N}{\mu b} \rceil}$.

The data space is thus partitioned into a set of equal-size intervals of length y. Similarly, a range query is a segment normalized to 1 which may span one or several contiguous intervals of length y. The fraction x of accessed pages can also be expressed in terms of the number of intervals of length y as follows:

$$x = \alpha y + \beta, \qquad where\ 0 \leq \beta < y$$

The formula establishes that the range query spans α intervals and a fraction β of an interval. The exact number of pages to be retrieved depends on the placement of the range query's boundaries over the data space. If the range query starts at an interval boundary, then the answer set will span $\lceil x/y \rceil$ intervals. The same is true if the range query starts anywhere within distance $(y - \beta)$ from the beginning of an interval. The probability for this case to occur is $(y - \beta)/y$.

On the other hand, if the range query starts anywhere within a distance β from the end of an interval, the answer set will span $\lceil x/y \rceil + 1$ intervals. Thus, the average number of retrieved pages is:
$\hat{N}_p = (y-\beta)/y * \lceil x/y \rceil + \beta/y * (\lceil x/y \rceil + 1) = \lceil x/y \rceil + \beta/y$
This can be approximated by: $\bar{N}_p = 1 + x/y$

Note that $\hat{N}_p = \bar{N}_p$ only when $\beta > 0$. When $\beta = 0$, we have $\hat{N}_p + 1 = \bar{N}_p$. Because the probability of $\beta = 0$ is negligible, we shall assume without loss of generality that $\hat{N}_p = \bar{N}_p$. Thus, $\bar{N}_p$ provides a simple formula to compute the average retrieval cost in the one-dimensional case.

Let us now consider the d-dimensional case, $(d > 1)$. The data space normalized to the unit cube $[0, 1]^d$ can be viewed as a multidimensional grid where each cell represents a page. Since data is uniformly distributed, the number of data pages remains unchanged from that of the one-dimensional case. Thus each cell occupies a volume y of the data space. Without loss of generality, the unit cube can be viewed as partitioned into the same number of intervals along each dimension. As a consequence, the width of a cell along each dimension is $(y)^{1/d}$.

Similarly, a range query can be viewed as a d-dimensional cube. Assuming that the range query retrieves a fraction x of the total number of records, this hypercube represents a volume x of the data space. In turn, the projection of this hypercube into each dimension has a width of $x^{1/d}$. Thus, by extrapolating the result obtained for the one-dimensional case to the d-dimensional case, the average number of pages accessed by the range query is: $\bar{N}_p^d = (1 + (x/y)^{1/d})^d$.

Our objective is to compute the growth rate of the number of accessed pages per range query as a function of the number of dimensions. For this purpose, it suffices to examine the magnitude of the ratio between the retrieval costs in one specific dimension, say dimension d, and the next dimension, $d + 1$. It is not difficult to show that as d increases, the ratio $\bar{N}_p^{d+1}/\bar{N}_p^d$ also increases. That is, $\frac{\bar{N}_p^2}{\bar{N}_p} < \frac{\bar{N}_p^3}{\bar{N}_p^2} < \ldots < \frac{\bar{N}_p^d}{\bar{N}_p^{d-1}}$.

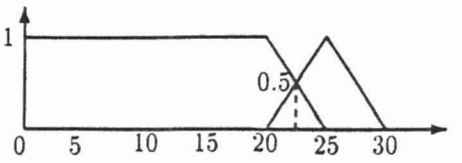

Figure 2: The truth value of "about 25 = young" is 0.5.

As a consequence, we have the following formula:

$$\bar{N}_p^d = \frac{\bar{N}_p^d}{\bar{N}_p^{d-1}} \cdot \frac{\bar{N}_p^{d-1}}{\bar{N}_p^{d-2}} \cdot \ldots \cdot \frac{\bar{N}_p^2}{\bar{N}_p} \cdot \bar{N}_p \;>\; (\frac{\bar{N}_p^2}{\bar{N}_p})^{d-1} \cdot \bar{N}_p$$

$$= (1 + \frac{2\sqrt{x/y}}{1 + x/y})^{d-1} \cdot (1 + x/y) \qquad (1)$$

It can also be shown that

$$\lim_{d \to \infty} \frac{(1 + (x/y)^{1/(d+1)})^{d+1}}{(1 + (x/y)^{1/d})^d} = 2 \qquad (2)$$

The above formula clearly indicates that the number of pages accessed increases exponentially as the dimension d increases. The growth rate is greater than 1 and converges to 2 as the number of dimensions increases.

5 Application of G-tree to Fuzzy Databases

Data in databases may not be accurate and users of a database may be interested in asking imprecise (fuzzy) queries. For example, an eye witness may see a person who committed a crime but is unable to describe the person accurately. In other words, the description of the characteristics of the suspect is fuzzy. The basic idea of fuzzy databases is that a fuzzy value in an attribute is represented as a function over an interval, say I_1. For example, the value "about 25" in the attribute "age" may be represented as a triangular function between 20 and 30 as shown in Fig. 2. The value of the function at x is the degree in which X matches "about 25" (e.g., the degree in which 26 matches "about 25" is 0.8). Similarly, a fuzzy value in a query is also represented by a function over an interval, say I_2. For example, the fuzzy value "young" is a function which has value 1 from age 0 to age 20 but decreases linearly from 20 to 25 when the value becomes 0 (shown in Fig. 2). The degree in which the data with interval I_1 satisfies the query with interval I_2 is determined as follows. If there is no intersection between I_1 and I_2, then the degree is 0; otherwise, the degree is the highest intersection point between the function associated with the data and that associated with the query. Figure 2 illustrates this idea.

The computation of the degrees of different tuples with a fuzzy query is carried out in two steps [18]. First, we identify the tuples which have positive degrees with the query. Those tuples have the associated intervals which have non-empty intersections with the interval

associated with the query. Second, for each non-empty intersection, the function associated with the tuple and that associated with the query will be used to compute the exact degree. In this paper, we concentrate on the first step. More precisely, we are interested in building an index, such that in response to a fuzzy query, those tuples whose intervals have non-empty intersections with the interval associated with the query will be retrieved for computation of the degrees.

5.1 Representation of Fuzzy Data

Fuzzy items are represented in a database as functions over intervals. For the purpose of indexing, only the intervals are of importance. Thus, in the following discussions, we only consider intervals. We represent interval $[a, b]$ associated with a fuzzy item by a point (a, b) in two-dimensional space. A precise item which has an associated interval $[c, c]$ for some value c is then represented by the point (c, c). In other words, an interval in fuzzy databases is transformed to a point in two-dimension space. A similar transformation was used in [4, 12] to analyze object oriented (rectangles) spatial methods. A fuzzy query with an associated interval $[d, e]$ is converted to the range query $0 \leq x \leq e, d \leq y \leq M$ in two-dimensional space, where M is the maximum value of the domain for the fuzzy attribute.

The following proposition indicates that the above representations are correct.

Proposition: The interval $[a, b]$ associated with a fuzzy data item D intersects with the interval $[d, e]$ associated with a fuzzy query Q if and only if the point (a, b) is retrieved by the range query $0 \leq x \leq e, d \leq y \leq M$.

5.2 Experimental Results

The experimental setup is as follows. 200,000 tuples are randomly generated; each tuple occupies 84 bytes; each page is of size 1k. There are three parameters in this experiment. (1) percentage of fuzzy data, p, which is the number of tuples having fuzzy values divided by the total number tuples; (2) data fuzzy degree, dfd, which is the average length of an interval associated with a fuzzy value in the database divided by M where the domain is from 0 to M; (3) query fuzzy degree, qfd, which is the length of an interval associated with a fuzzy query divided by M. Twenty queries with the same fuzzy degree are randomly generated and the results to be presented are averaged over the 20 queries.

The three parameters are varied as follows: p varies from 10% to 50% with increment of 10%; dfd takes one of the three values: 1%, 5% and 10%; and qfd takes one of the three values: 0%, 1% and 10%.

Representative sets of experimental results are shown in tables 6, 7, 8 and 9. Columns *t_io*, *i_io* and *d_io* are the total number of pages, the number of index pages and the number of data pages accessed respectively. Columns $\#t$ and $\%t$ represent the total number of tuples and the percentage of tuples retrieved respectively. The last column eff represents the efficiency of the index which is defined as eff $= \frac{d}{t\_io}$, where d is the minimum number of data pages to contain the answer.

p	t_io	i_io	d_io	#t	%t	eff(%)
10	253.7	112.4	141.3	869.8	0.4	28.6
20	355.5	106.6	248.9	1748.8	0.87	41.0
30	469.3	108.4	360.9	2623.8	1.31	46.5
40	569.7	97.1	472.6	3483.7	1.74	51.0
50	676.6	95.8	580.8	4317.3	2.16	53.2

Table 6: Precise queries on fuzzy data ($dfd = 5\%$)

When the percentage of fuzzy data is 10% and the query is precise (the first row of Table 6), the minimum number of pages to contain the answer to the query is 73. But the actual number of pages accessed is 254 consisting of 113 index pages and 141 data pages. As a result, the efficiency is only 28.6%. When the percentage of fuzzy data increases from 10% to 50% (Table 6), the efficiency increases to 53.2%. When qfd increases from 0% to 10%, the efficiency increases from 53.2% to 63.3% (the last rows of tables 6, 7 and 8). When dfd increases from 1% to 10%, the efficiency increases as indicated in table 9.

In summary, the G-tree structure is less efficient for precise queries on relatively precise data; its performance for imprecise queries on relatively imprecise data is acceptable. We note that the G-tree structure is also applicable to temporal databases where each tuple is associated with a time interval. A temporal query is also associated with a time interval. A typical query retrieves all those tuples whose time intervals intersect with the time interval of the query. If the length of the interval associated with each tuple is greater than zero, then this corresponds to 100% fuzzy data in fuzzy databases. This yields higher efficiency. Thus, this structure offers a new alternative to existing temporal access methods [1, 3, 16]. However, the performance for precise temporal queries (i.e., find all the intervals that overlap a time instance) will be worse than that for interval queries.

6 Conclusion

We implemented the G-tree spatial access method and provided a comprehensive empirical analysis of its performance on space utilization, insertions, deletions, point queries, range queries and partial queries. The average bucket utilization for various data distributions

p	t_io	i_io	d_io	#t	%t	eff(%)
10	554.7	112.4	442.3	3337.9	1.67	50.3
20	620.8	106.6	514.2	3892.3	1.95	52.4
30	721.9	108.4	613.5	4702.0	2.35	54.3
40	816.2	97.1	719.1	5509.8	2.75	56.2
50	923.2	95.8	827.4	6308.1	3.15	56.8

Table 7: Fuzzy queries ($qfd = 1\%$) on fuzzy data ($dfd = 5\%$)

p	t_io	i_io	d_io	#t	%t	eff(%)
10	3261.4	124.7	3136.7	25955.2	12.98	66.2
20	3077.8	118.0	2959.8	23658.5	11.83	64.1
30	3046.8	119.4	2927.4	23339.4	11.67	63.7
40	3115.7	108.0	3007.7	23750.5	11.88	63.7
50	3217.7	106.4	3111.3	24388.6	12.19	63.3

Table 8: Fuzzy queries (qfd = 10%) on fuzzy data (dfd = 5%)

qfd	dfd	t_io	i_io	d_io	#t	%t	eff(%)
0	1	184	105	79	410	0.2	18.6
	5	356	107	249	1749	0.9	41.0
	10	536	105	431	3145	1.6	48.8
1	1	457	105	352	2549	1.3	46.7
	5	621	107	514	3892	1.9	52.4
	10	809	106	703	5369	2.7	55.2
10	1	2934	116	2818	22119	11.1	63.0
	5	3078	118	2960	23658	11.8	64.1
	10	3261	117	3144	25286	12.6	64.5

Table 9: Queries on fuzzy data with variable fuzzy degree (p = 20%)

is around 69%, which is consistent with the result previously reported in [7], and in other similar structures [5, 11, 14]. The average number of disk accesses per point query is approximately the height of G-tree plus one. For an insertion, the cost is approximately the height of G-tree plus two. The same is true for deletions. These results (for insertions, deletions and point queries) are generally independent of the data distributions. For range queries, the average number of disk accesses changes with respect to the data distributions. The changes are within 45% for the 2-dimensional case. The number of disk accesses increases sub-linearly as the region size of partial/range queries increases. When the dimension is increased to 3, the relative increase in the number of disk accesses is very substantial for range queries retrieving a small percentage of data, but not so substantially for queries retrieving much more data. G-tree performs better than BHT [8] for queries retrieving substantial amount of data while its performance for queries retrieving very small amount of data is not as good as those of the other methods [8].

We design a model to study performance of d-dimensional searching. A simple formula which gives the number of data pages searched is derived. This is independent of the index structure and therefore the formula is applicable to any d-dimensional searching scheme. As a consequence of this formula, it is shown that as d increases, the number of data pages increases exponentially. The growth rate is greater than 1 and converges to 2 as the number of dimensions increases.

We apply the G-tree structure to fuzzy databases and find that it has good performance for fuzzy queries on substantial fuzzy data. However, the structure is less efficient for answering precise queries on relatively precise data. We note that the same structure can be applied to temporal databases. If each tuple has an associated interval with length > 0, then it corresponds to a fuzzy tuple and the retrieval performance in such an environment is acceptable.

References

[1] Ahn, I. and Snodgrass, R., *Partitioned Storage for Temporal Databases*, Information Systems, 13(4), 1988.

[2] Burkhard., W., *Interpolation-based Index Maintenance*, Proc. ACM PODS, 1983, pp76-85.

[3] Elmasri, R., Wuu, G.T.J. and Kim, Y.J., *The Time Index: An Access Structure for Temporal Data*, Proc. 16th VLDB Conf., Australia, 1990, pp1-12.

[4] Faloutsos, C., Sellis, T. and Roussopoulos, N., *Analysis of Object Oriented Spatial Access Methods*, Proc. ACM SIGMOD Conf., San Francisco, 1987.

[5] Freeston, M., *The BANG file: a new kind of grid file*, Proc. ACM SIGMOD Conf., San Francisco, 1987.

[6] Hinrichs, K., *Implementation of the Grid File:Design Concepts and Experiences*, BIT 25, 1985, pp569-592.

[7] Kumar, Akhil, *G-tree: A New Data Structure for Organizing Multidimensional Data*, IEEE Trans. Knowledge Data Eng., Vol. 6, No. 2, April 1994, pp341-347.

[8] Kriegel, H.-P., Schiwietz, M., Schneider, R., Seeger, B., *Performance Comparison of Point and Spatial Access Methods*, Proc. 1st Symp. on the Design of Large Spatial Databases, 1989, pp.89-114.

[9] Lomet, D. and Salzberg, B., *The hB-tree: A Robust multiattribute Search Structure*, Proc. of IEEE Data Engineering Conf., Feb. 1989.

[10] Nievergelt, J, Hinterberger, H., and Sevcik, K.C., *The Grid File: An Adaptable, Symmetric Multikey File Structure*, ACM TODS, vol.9, no.1, Mar. 1984, pp38-71.

[11] Orenstein, J.A. and T.H. Merrett, *A Class of Data Structures for Associative Searching*, Proc. of ACM PODS, 1984, pp. 181-190.

[12] Orenstein, J.A., *Spatial Query Processing in an Object Oriented Database System*, Proc. of ACM-SIGMOD Conf., Washington, DC, May, 1986.

[13] Ouksel, M., and Scheuermann, P., *Storage Mappings For Multidimensional Linear Dynamic Hashing*, Proc. ACM PODS, 1983, pp90-105.

[14] Ouksel, M., *The Interpolation-based grid file*, Proc. ACM PODS, 1985, pp20-27.

[15] Ouksel, M., Scheuermann, P., *Implicit Data Structures for Linear Hashing Schemes*, Information Processing Letters 29, 1988, pp183-189.

[16] Rotem, D. and Segev, A., *Physical Organization of Temporal Data*, Proc. IEEE Data Engineering Conf., 1987.

[17] Salzberg, B., and Lomet, D., *Spatial Database Access Methods*, SIGMOD RECORD, vol 20, no. 3, Sep. 1991, pp6-15.

[18] Yang, Q. Liu,C., Wu,J., Yu,C. Dao,S. and Nakajima,H. *Efficiency Processing of nested fuzzy SQL queries*, IEEE Data Engineering Conf., 1995.