Multiattribute Hashing Using Gray Codes.

Christos Faloutsos

Dept of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Multiattribute hashing and its variations have been proposed for partial match and range queries in the past The main idea is that each record yields a bit-string $\vec{b}$ ("record signature"), according to the values of its attributes The binary value $(\vec{b})_2$ of this string decides the bucket that the record is stored In this paper we propose to use Gray codes instead of binary codes, in order to map record signatures to buckets In Gray codes, successive codewords differ in the value of exactly one bit position, thus, successive buckets hold records with similar record signatures The proposed method achieves better clustering of similar records and avoids some of the (expensive) random disk accesses, replacing them with sequential ones
We develop a mathematical model, derive formulas giving the average performance of both methods and show that the proposed method achieves 0% - 50% relative savings over the binary codes We also discuss how Gray codes could be applied to some retrieval methods designed for range queries, such as the grid file [Nievergelt84a] and the approach based on the so-called z-ordering [Orenstein84a]

1. Introduction.

Access methods for secondary key retrieval have attracted much interest The problem is stated as follows Given a file F and information about the queries (type of queries, frequencies), design an efficient file structure to handle these queries The file F is a collection of ordered k-tuples (records), each consisting of k keys or attribute values Attributes are denoted by $A_1, A_2, \ldots, A_k$ Some common classes of queries are, in increasing order of complexity

1) Exact match queries The query specifies all the attribute values of the desired record, eg

$$A_1=v_1 \wedge A_2=v_2 \wedge \ldots \wedge A_k=v_k$$

2) Partial match queries Only some of the attribute values are specified, eg

$$A_3=v_3 \wedge A_5=v_5$$

3) Range queries Ranges for some or all of the attributes are specified, eg

$$(l_2 \leq A_2 \leq u_2) \wedge (l_3 \leq A_3 \leq u_3)$$

4) General Boolean queries, eg

$$\left[(A_1=v_1) \vee (l_2 \leq A_2 \leq u_2)\right] \wedge (A_4=v_4)$$

We focus on partial match queries and discuss range queries in section 4 Some of the most prominent methods are inverted files [Cardenas75a], multidimensional tree-structures (k-d trees [Bentley75a], k-d B-trees [Robinson81a]), grid file [Nievergelt84a], and finally, multiattribute hashing [Rothnie74a], which we shall focus on in this paper

According to multiattribute hashing, each record yields a bit-string $\vec{b}$ of size n ("record signature"), by applying a hashing function to each attribute value and combining (eg , concatenating) the binary representations of the hash values ("attribute signatures") The binary value $(\vec{b})_2$ decides the bucket that the record is stored Thus, records with the specified combination of attribute values are not scattered randomly within

the storage space, instead, they are contained in a well-defined set of buckets, resulting in fewer bucket (disk) accesses

The advantages of multiattribute hashing are

1) The automatic "clustering" of similar records as described above, that saves disks accesses
2) The address of a record is determined immediately from the record itself, without the need of consulting an inverted index or traversing a tree structure Maintenance of an index under many insertions and deletion is a time consuming task, which is avoided with multiattribute hashing
3) The method does not require merging of pointer lists on partial match queries Moreover, the amount of work to answer such a query decreases (usually exponentially) with the number of attribute values specified in the query, in contrast, for the inverted file method the work increases (more pointer lists to merge)

The improvement we propose in this work is to store consecutively buckets with similar hash codes The similarity is measured by the number of bit positions that the hash codes differ (Hamming distance) Using Gray codes [Gray53a], consecutive buckets differ only in one bit position in their hash codes We derive formulas for the expected savings we can achieve with this method, and we show that, for any partial match query, the savings range from 0% to 50% The only overhead of the method is the computation to transform Gray codes to binary codes

We also note that the idea of using Gray codes can be applied indistinguishably to any variation of multiattribute hashing, as well as to other secondary-key access methods, such as the grid file [Nievergelt84a] or the approach based on the so-called "z-ordering" [Orenstein84a], which are geared towards range queries

The structure of the paper is as follows In section 2 we present a brief description of multiattribute hashing, give an example of the proposed idea and provide some background information on Gray codes In section 3 we present the performance analysis of the proposed method In section 4 we discuss improvements and additional applications of the method In section 5 we list the conclusions and the contributions of this work

2. Background: Multiattribute Hashing - Gray codes.

The idea of multiattribute hashing is best illustrated with an example Consider a file with employee records, with three attributes A_1=NAME, A_2=AGE and A_3=SALARY For each attribute we have a hashing function, $h_{NAME}(\,)$, $h_{AGE}(\,)$ and $h_{SALARY}(\,)$, which maps the corresponding attribute values to bit strings of a prespecified length (2, 1 and 1 respectively in our example) The bit string that corresponds to an attribute value will be called **signature** of this attribute value The length of a signature affects proportionally the performance upon queries on the specific attribute Consider the following hashing functions

$$h_{NAME}(x)=\begin{cases}00 & \text{if} \quad x<\text{"}E\text{"}\\ 01 & \text{if "}E\text{"} \le x<\text{"}L\text{"}\\ 10 & \text{if "}L\text{"} \le x<\text{"}S\text{"}\\ 11 & \text{if "}S\text{"} \le x\end{cases}$$

$$h_{AGE}(y)=\begin{cases}0 & \text{if} \quad y \le 35\\ 1 & \text{if } 35<y\end{cases}$$

$$h_{SALARY}(w)=\begin{cases}0 & \text{if} \quad w \le 25000\\ 1 & \text{if } 25000<w\end{cases}$$

For partial match queries, hashing functions need not be order preserving Concatenating the signatures of the attribute values of a record results in a bit string, which we call **record signature.** It is the record signature that decides which bucket to store a record in In our example, the record

("Smith", 40, 22000)

yields the following attribute signatures h_{NAME}("Smith")="11", $h_{AGE}(40)$="1", $h_{SALARY}(22000)$="0" and the record signature is "1110" The record will be stored in the 14-th bucket ($(14)_{10}=(1110)_2$), as shown in Figure 1 There, for the sake of the example, each bucket holds 2 records In real applications the bucket capacity would be larger Overflows are handled with any of the known techniques, eg separate chaining In the rest of the paper, we assume that the overflow probability is negligible

We define as **characteristic signature** of a bucket the (common) record signature of the records in this bucket For example, in Figure 1 the characteristic signature of bucket #14 is the binary string 1110 In general, the numeric value of the characteristic signature of a bucket is the offset of this bucket from the beginning of the hash table

Searching for a partial match query is performed as follows Consider the query

SALARY=20000 $\wedge$ AGE=50

The qualifying records (if any) will have signature of the form ??10, where "?" stands for the "don't care" character The buckets that hold such records are buckets #2, #6, #10 and #14

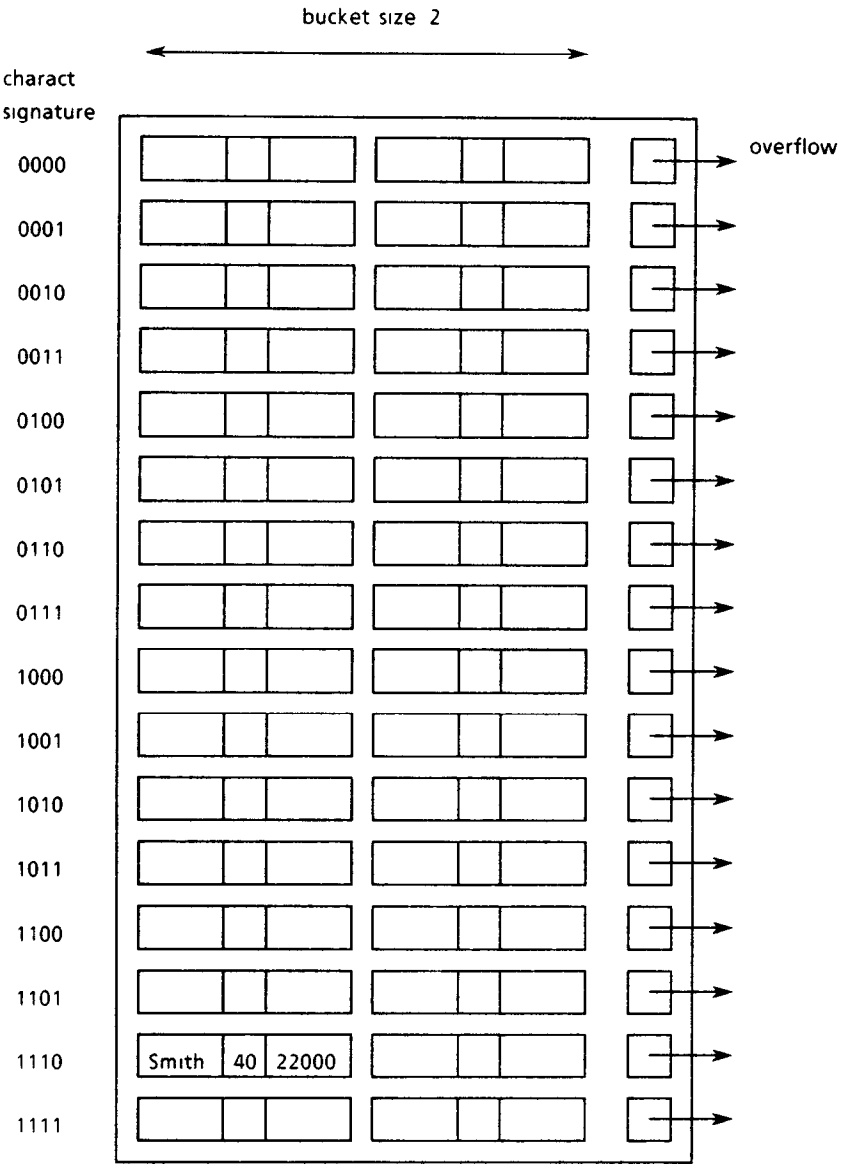

Figure 1
Illustration of multiattribute hashing
Bucket size = 2 records

Some definitions A **query signature** γ is a string of n characters from the set { "0", "1", "?" } A **q-bit query** is a query whose signature specifies q bit positions, therefore, it contains $n-q$ "don't care" characters "?"

The initial version of multiattribute hashing was proposed by Rothnie and Lozano [Rothnie74a] Improvements and variations were published by Rivest [Rivest76a], Aho and Ullman [Aho79a], Lloyd [Lloyd80a], Lloyd and Ramamohanarao [Lloyd82a] and Ramamohanarao, Lloyd and Thom [Ramamohanarao83a]

The improvement we are proposing can be applied to any of the above methods and it was motivated by the following observation Neighboring buckets may have drastically different characteristic signatures (see Figure 1), the extreme being buckets #7 and #8, which differ in all four bit positions The result is that quali-

fying records may be far apart, requiring many random accesses on the disk As a remedy, we propose to use Gray codes [Gray53a] to map characteristic signatures to bucket addresses By definition, *successive codewords in Gray codes differ in one only bit position* Figure 2 shows the codewords of a 4-bit Gray code, along with their order in binary and decimal The illustrated code belongs to the class of the so-called **binary reflected Gray codes** which will be examined in more detail A compact representation of a Gray code is by its **transition sequence,** which is defined as the ordered list of bit positions (numbered from right to left) that change as we go from one codeword to its successor For example, the transition sequence of the Gray code of Figure 2 is (1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1)

decimal	Gray	binary
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Figure 2
Illustration of the 4-bit binary reflected Gray code

To illustrate that the Gray codes achieve savings in random accesses, consider the query signature "??10" of the previous example The buckets that qualify in the Gray code method are buckets #3, #4, #11 and #12, which require only 2 random accesses, instead of the 4 random accesses that the binary method requires In the next section we examine the performance gains in more detail Here we briefly discuss how to generate Gray codes and how to find the order (or "position", "offset", "index") of a given codeword in the code The forthcoming definitions and formulas are from [Reingold77a]

A n-bit Gray code $G(n)$ can be represented as a $n * 2^n$ binary matrix, with each row being a codeword (See Figure 2, for $G(4)$) If

$$G(n) = \begin{bmatrix} G_{n,0} \\ G_{n,1} \\ \\ G_{n,2^n-1} \end{bmatrix}$$

is a n-bit Gray code, then obviously the code

$$G(n+1) = \begin{bmatrix} 0G_{n,0} \\ 0G_{n,1} \\ \\ 0G_{n,2^n-1} \\ 1G_{n,2^n-1} \\ \\ 1G_{n,1} \\ 1G_{n,0} \end{bmatrix}$$

is a $n+1$-bit Gray code Applying this technique recursively we have the **binary reflected Gray codes,** with the 1-bit code trivially defined as

$$G(1) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Informally, in the binary-matrix representation of a binary reflected Gray code, the lower half of the matrix is the mirror image of the upper half, if the leftmost column is neglected For the rest of the paper, unless otherwise stated, the term "Gray code" will imply "binary reflected Gray code" The notation $(.)_G$ will denote the **order** of the parenthesized binary string in the binary reflected Gray code For example, $(1110)_G = 11$ $(=(1011)_2)$, as can be verified from Figure 2 The conversion of a Gray-codeword to its order is needed in our application For example, to insert the record with signature "1110", we have to calculate the order $(1110)_G$, which will be the address of the bucket that the record should be stored in The formulas presented in [Reingold77a] convert the n-bit Gray-codeword $(g_n\ g_{n-1}\ \ \ g_1)$ to its order in binary $(b_n\ b_{n-1}\ \ \ b_1 b_0)$ The formulas are

$$b_n = 0 \qquad (2.1)$$

$$b_j = \sum_{m=j+1}^{n} g_m \pmod 2 \quad 0 \le j < n \qquad (2.2)$$

The conversion of Gray codewords to their order is the *only* overhead that the proposed method requires Notice that formula (2.2) is linear on n and that its computation is performed with CPU speed Therefore, the overhead of the method is small, compared to the savings in I/O time it can achieve Next we examine the performance gain in more detail

3. Performance Analysis.

The measure of performance will be the number of "clusters" C that the qualifying buckets form in resolving a query A cluster for a given query is a set of consecutive, qualifying buckets For example, the query signature "??10" retrieves 4 (one-bucket) clusters when we use binary codes, and 2 clusters when we use Gray codes (see Figure 3) The number of clusters is the number of random accesses required to resolve a query Since random accesses cost more than sequential ones, the number of clusters is a reasonable performance indicator

decimal	Gray	binary
0	0000	0000
1	0001	0001
2	0011	0010 •
3	0010 •	0011
4	0110 •	0100
5	0111	0101
6	0101	0110 •
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010 •
11	1110 •	1011
12	1010 •	1100
13	1011	1101
14	1001	1110 •
15	1000	1111

Figure 3
Illustration of the improved clustering of Gray codes
Query signature "??10"
Qualifying buckets are marked with a bullet

The assumptions we make for the environment are

1) Each bucket occupies exactly one disk block (page)
2) The buckets of our hash table are physically stored consecutively
3) The probability of overflow is neglected Large bucket size, good hashing functions and no correlation among attributes should result in small overflow probability

We shall compare the binary versus the Gray code method of assigning characteristic signatures to buckets Notice that in both methods, the total number of disk accesses (= number of buckets examined) is the same for the same query, however, the number of clusters C (= random accesses) should be smaller for the Gray codes

The setting is as follows We have a hash table with 2^n buckets The users issue partial match queries, which are translated to query signatures We want the number of clusters that each query will find under each method Since the number of distinct query signatures grows combinatorially with n, we are interested in the number of clusters that a query finds on the *average* More detailed, we group the queries according to the number of bits q specified in their query signature Suppose that all the q-bit query signatures are equiprobable, for a fixed q Then the problem is to find the average number of clusters $\overline{C}_B(n,q)$ and $\overline{C}_G(n,q)$ that a q-bit query signature will encounter in the binary and Gray code method respectively The details of the derivations are presented in Appendices 1 and 2 The results are

$$\overline{C}_B(n,q)=\frac{\sum_{i=1}^{n-q+1} comb(n-i,q-1)2^{n-q-i+1}}{comb(n,q)} \quad (3.1)$$

$$\overline{C}_G(n,q)=\frac{1}{(2^q\, comb(n,q))}*$$

$$\left[comb(n,q)+(2^n-1)comb(n-1,q-1)\right] \quad (3.2)$$

$$\overline{Gain}(n,q)\approx\frac{n-q}{2(n-1)}$$

where

$$\overline{Gain}(n,q)\triangleq\frac{\overline{C}_B(n,q)-\overline{C}_G(n,q)}{\overline{C}_B(n,q)}$$

is the relative performance gain of the proposed method and

$$comb(n,m)\triangleq\frac{n!}{m!(n-m)!}$$

is the number of n-choose-m combinations

Table 3.1 shows the relative savings for $n=10$ and $1\le q\le 10$ Tracing manually all the possible query signatures for $n=4$ bits, we observed that, for an individual query, the Gray code method never does worse than the binary method This is not a coincidence However, the proof is lengthy [Faloutsos85a] and it will not repeated here The main result is

$$C_B(n,\vec{q})\ge C_G(n,\vec{q})\ge\frac{C_B(n,\vec{q})}{2}\quad\forall\vec{q} \quad (3.3)$$

where $\vec{q}$ is a query signature and $C_B(n,\vec{q})$, $C_G(n,\vec{q})$ are the numbers of clusters for this query under binary and Gray codes respectively

query bits specified q	avg # of clusters per query (Gray codes)	avg # of clusters per query (binary)	savings per cent
1	51 65	102 3	49 5112
2	51 4	91 0444	43 5441
3	38 4875	61 8583	37 7812
4	25 6375	37 7952	32 1674
5	16 0156	21 8373	26 6593
6	9 60625	12 1952	21 2295
7	5 60234	6 65833	15 8597
8	3 20078	3 57778	10 5372
9	1 8002	1 9	5 25288
10	1	1	0

Table 3 1
Comparison of the expected number of clusters per query, for q-bit queries $1 \leq q \leq n = 10$

Symbol	Definition
n	size of record signature (bits)
$\vec{q}$	a query signature
q	number of bit positions specified in the query signature
$\vec{q}_i$	a query signature, whose rightmost bit-position specified is i ($i-1$ trailing "?"'s)
$\overline{C}_G(n,q)$	Expected number of clusters per query (Gray codes, q-bit queries)
$C_{G,tot}(n,q)$	Total number of clusters for all the q-bit queries (Gray codes)
$C_G(n,\vec{q})$	Number of clusters for a specific query $\vec{q}$ (Gray codes)
$\overline{C}_B(n,q)$	Expected number of clusters per query (binary codes, q-bit queries)
$C_{B,tot}(n,q)$	Total number of clusters for all q-bit queries (binary codes)
$C_B(n,\vec{q})$	Number of clusters for a specific query $\vec{q}$ (binary codes)

Table 3 2
Symbol definitions

Eq (3 3) implies that the relative savings of the Gray code method for any query signature $\vec{q}$ lie in the interval (0%, 50%) In [Faloutsos85a] we also show that the maximum saving (50%) is achieved by query signatures of the form

$$\vec{q}_{opt} = [01?]^x \; ?1\,0^y \; ?^z \quad x,y,z \geq 0$$

where exponentiation of a symbol implies repetition, eg , "01^3" = "0111" Square brackets imply arbitrary choice of the enclosed symbols

4. Discussion.

Here we examine extendible versions of multiattribute hashing as well as methods for handling range queries In both cases, we show how Gray codes can be applied and illustrate (without proof) the performance improvement

4.1. Extendibility.

Many extendible primary key hashing methods are known, including dynamic hashing [Larson78a], extendible hashing [Fagin79a], linear hashing [Litwin80a], spiral hashing [Martin79a], linear hashing with partial expansions [Larson82a] Two of them, namely extendible hashing, and linear hashing were recently applied for partial match retrieval [Lloyd82a] and [Ramamohanarao83a], without using Gray codes We shall only describe how extendible hashing can cooperate with Gray codes The reason for our choice is that extendible hashing shares some ideas with the grid file method, which we discuss later, in the subsection for range queries The modification for linear hashing is left to the reader

First, a brief description of the multiattribute version of extendible hashing (see Figures 4-5) Consider records with k attributes, A_1, A_2, A_k Consider also n bit record signatures, n_1 of which are derived from attribute A_1, n_2 from A_2, , n_k from A_k In the example of Figure 4, only $d=2$ of these n bits are used According to extendible hashing, there is a directory with 2^d entries, each entry containing a pointer to the bucket where the corresponding records are stored d is the **depth** of the directory Bucket #1 has **local depth** $d'=1$, since it contains records with record signatures agreeing up to the 1-st (leftmost) bit Buckets #2 and #3 have local depth $d'=2$ The main idea in extendible hashing is illustrated when insertion of a new record causes a bucket with local depth $d'=d$ to overflow For example, assume that bucket #3 of Figure 4 overflows The following actions take place

- A new bucket (#4) is obtained
- One more bit out of the n bits of the record signatures is used to determine which of the records of the overflowing bucket (#3), will be sent to the new bucket (denoted by dotted circles in Figure 5)
- The directory doubles in size, and its entries are filled as shown in Figure 5

The proposed modification is to use Gray codes, in order to assign record signatures to directory entries Figures 6 and 7 illustrate the situation before and after the overflow of bucket #3, when Gray codes are used

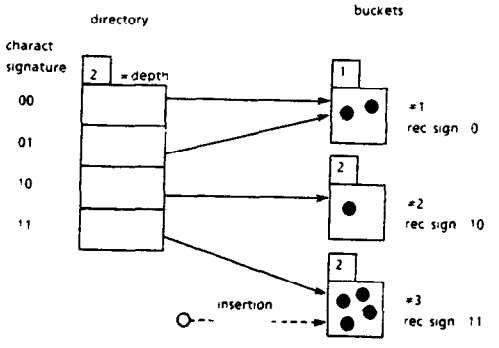

Figure 4
Illustration of extendible hashing

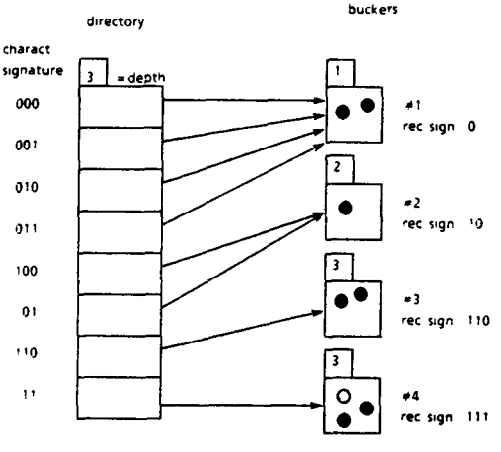

Figure 5
Directory after the split of bucket #3
Dotted circles denote records
that moved after the split

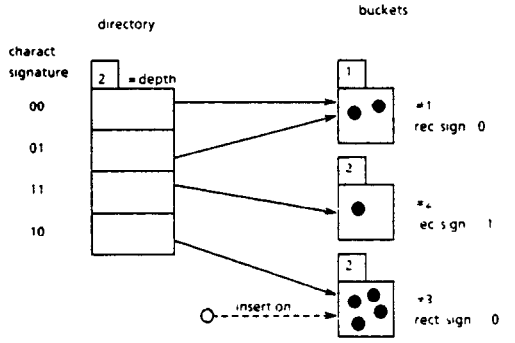

Figure 6
Extendible hashing using Gray codes

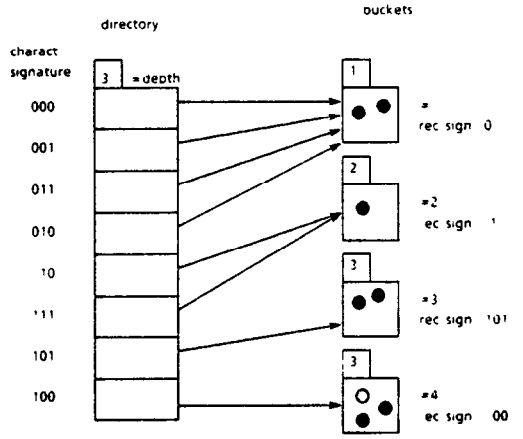

Figure 7
Extendible hashing with Gray codes,
after the split of bucket #3

Notice that buckets are not stored consecutively, since they are dynamically allocated However, Gray codes can still save I/O time for partial match queries, if the directory size is large For a directory depth of d =20 bits, the directory has 2^{20}=1M entries and it can not reside in main memory Gray codes achieve better clustering of the directory entries, saving accesses to the directory Figure 8 gives an example of this claim Assuming that 3 directory entries fit in one block, the query signature "??1?" retrieves 4 blocks under Gray codes, instead of 6 blocks under binary codes

decimal	Gray	binary
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

Figure 8
Illustration of Gray codes applied to the directory
of extendible hashing (3 entries per block)
The query signature "??1?" retrieves 4 (6) blocks
of the directory under Gray (binary) codes

4.2. Range queries

Gray codes might improve the performance of methods designed for range queries Grid file [Nievergelt84a] is such a method Like the multiattribute version of extendible hashing, it uses a directory to store pointers to buckets Based on the previous discussion and example of Figure 8, Gray codes could reduce the amount of I/O required to access the grid directory

A generalized approach to secondary key retrieval, that may be improved with Gray codes, was suggested by Orenstein and Merrett [Orenstein84a] The main ideas are a) to use order preserving hashing functions (in their paper they use one-to-one encoding of the attribute values into binary numbers) b) to create the

record signature by **interleaving** bits from the attribute signatures, according to a **shuffling function.** For example, under the shuffling function S_1=(2, 1, 2, 1), the record signature will be formed (from left to right) by taking the leftmost bit from the signature of A_2, then the leftmost bit of A_1, then the second leftmost bit from A_2 and finally the second leftmost from A_1 Thus, a record with signatures "00" and "11" for A_1 and A_2 respectively, will have the signature $shuffle(S_1$," 00" ," 11")= "1010" Thus, tuples can be ordered according to the value of their signatures, called **z-value** from now on Clearly, the induced ordering (so called **z-ordering**) depends on the shuffling function The main contribution of the above work is that, using the z-value of a record as a primary key, any primary-key file structure is immediately transformed to a secondary-key file structure The shuffling helps to assign similar z-values to similar records, thus resulting in storing these records in the same or neighboring disk blocks (pages) Therefore, a range query will retrieve fewer blocks

An example of z-ordering Consider a relation with k=2 attributes and domain(A_1)=domain(A_2) ={0,1,2,3} There are 16 possible distinct tuples, which are depicted as points in the 2-dimensional space (circles in Figures 9-11) The values of A_1 and A_2 are encoded in binary Figure 9 illustrates the z-ordering induced by the shuffling function S_0=(1, 1, 2, 2) The arrows on the traversal path lead to increasing z-values S_0 is the simple concatenation of the binary encodings of A_1 and A_2 Notice that the above ordering clearly favors A_1 over A_2 record with the same value for A_1 differ slightly in their z-values, which is not true for A_2 Figure 10 illustrates the z-ordering for the shuffling function S_1=(2, 1, 2, 1) Notice the more symmetric handling of the two attributes

We think that similar records can be brought even closer, thereby improving performance, by using Gray codes to encode the attributes signatures, as well as to compare the record signatures (z-ordering) Figure 11 illustrates the z-ordering under Gray codes, with shuffling function S_1= (2, 1, 2, 1) Notice that the values of the attributes along the axes are encoded in Gray codes The trail consists only of horizontal and vertical segments, avoiding the diagonal "jumps" of Figures 9 and 10, we conjecture that diagonal jumps are undesirable, because they indicate assignment of successive z-values to records without common attribute values

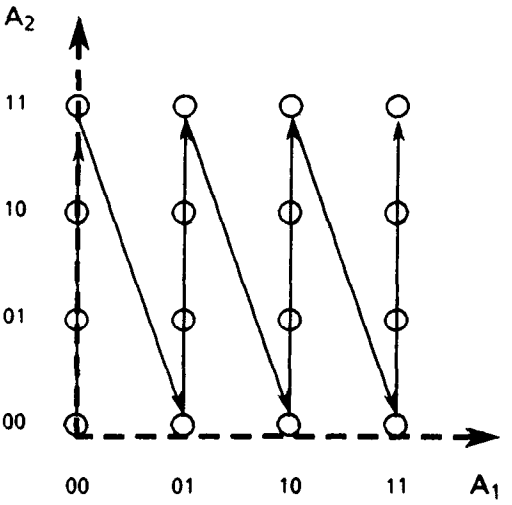

Figure 9
Illustration of z-ordering
Shuffling function (1,1,2,2)

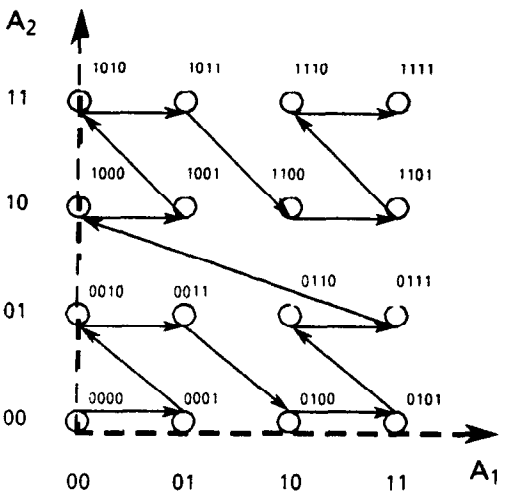

Figure 10
Illustration of z-ordering
Shuffling function (2, 1, 2, 1)

5. Conclusions

The main conclusion is that Gray codes can be used in any variation of multiattribute hashing, to save random disk accesses The penalty is a small computational overhead, namely the computation of the order of the given codeword in the Gray code Moreover, Gray codes can be used in distance-preserving mappings, to map a k-dimensional space to an one-dimension space Such mappings can be used to design the physical layout of a file with k attributes

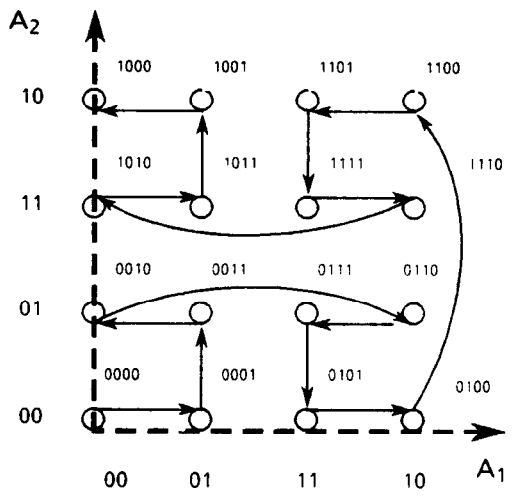

Figure 11
Illustration of z-ordering with Gray codes
Shuffling function (2, 1, 2, 1)

(k-dimensional space) on a 1-dimensional storage medium (disk) Distance-preserving mappings are useful for range queries The contributions of this work are

1) Proposal of using Gray codes in multiattribute hashing and related methods
2) Derivation of formulas for the average performance (= number of clusters) of both binary and Gray codes
3) Computation of the savings that Gray codes can achieve (0% - 50% relative savings, for any query signature)

Future research can deal with

- The design of "symmetric" Gray codes Binary reflected Gray codes favor queries that specify bits on the leftmost positions of the record signature The reason is that the smaller the number, the more often it appears in the transition sequence of the binary reflected Gray codes This variance in performance may be undesirable Gilbert [Gilbert58a] provides a list of all the possible Gray codes, for sizes $n \leq 4$ Some of them are "symmetric", in the sense that all numbers appear equally often in the transition sequence Systematic production of "symmetric" Gray codes for $n > 4$ is an interesting problem

  More detailed study of the application of Gray codes in the grid file method and in the z-ordering approach

Appendix 1. Number of clusters for the binary codes.

Here we calculate the average number of clusters $\overline{C}_B(n,q)$, $\overline{C}_G(n,q)$, for a q-bit query signature, under binary and Gray codes respectively

Consider a q-bit query and let s be the position of the least significant (=rightmost) bit of the q bits specified ($1 \leq s \leq n$) Let $\vec{q}_s$ be the query signature and let $C_B(n,\vec{q}_s)$ be the number of clusters that $\vec{q}_s$ retrieves Then, we have

$$C_B(n,\vec{q}_s) = 2^{n-q-s+1} \qquad \text{(A1 1)}$$

Our goal is to calculate the total $C_{B,tot}(n,q)$ of all the clusters for all the possible q-bit queries This is

$$C_{B,tot}(n,q) = 2^s \sum_{s=1}^{n-q+1} comb(n-s,q-1) C_B(n,\vec{q}_s)$$

or

$$C_{B,tot}(n,q) = 2^s * \sum_{s=1}^{n-q+1} comb(n-s,q-1) 2^{n-q-s+1} \qquad \text{(A1 2)}$$

The term $comb(n-s,q-1)$ is the number of ways of choosing the remaining $q-1$ bits in the query among the $n-s$ bits since s is the rightmost bit specified in the query The term 2^s is the number of different query signatures, for each set of q out of n bit positions specified in a query Since the total number of q-bit query signatures is

$$Q(n,q) = 2^s comb(n,q) \qquad \text{(A1 3)}$$

and the query signatures are assumed equiprobable, then the average number of clusters $\overline{C}_B(n,q)$ for q-bit queries is

$$\overline{C}_B(n,q) = \frac{1}{comb(n,q)} 2^{n-q} * \sum_{s=1}^{n-q+1} comb(n-s,q-1) 2^{-s+1} \qquad \text{(A1 4)}$$

Appendix 2. Number of clusters for Gray codes.

For the case of Gray codes, we base the calculations on the property that successive codewords differ in one and only one bit position Since this is the definition for the Gray codes in general (and not only for the binary reflected ones), *the forthcoming results apply to all the Gray codes*

An n-bit Gray code obviously consists of 2^n codewords Moving from the j-th codeword to the $j+1$-th, a change in one bit position happens In total, there are $2^n - 1$ such changes If these changes didn't

occur, then the Gray code would be a $2^n \times n$ bit array, full of zeros, containing $comb(n,q)$ q-bit clusters

A q-bit cluster is a set of consecutive codewords, which agree on the values of q specific bit positions Each change between successive codewords creates $comb(n-1,q-1)$ new q-bit clusters Since the total number of changes is 2^n-1, the total number of q-bit clusters is

$$C_{G,tot}(n,q) = comb(n,q) + (2^n-1)comb(n-1,q-1) \quad \text{(A2 1)}$$

and the expected number of clusters per q-bit query is

$$\overline{C}_G(n,q) = \frac{1}{2^q\, comb(n,q)} * \left[comb(n,q) + (2^n-1)comb(n-1,q-1) \right] \quad \text{(A2 2)}$$

The expected relative performance gain $\overline{Gain}(n,q)$ is defined as

$$\overline{Gain}(n,q) \triangleq 1 - \frac{\overline{C}_G(n,q)}{\overline{C}_B(n,q)}$$

or

$$\overline{Gain}(n,q) = 1 - \frac{1 + 2^{-1}\frac{n-q}{n-1} + 2^{-2}\frac{(n-q-1)(n-q)}{(n-1)(n-2)} + \cdots}{1 + 2^{-n}\frac{n-q}{q}}$$

Neglecting the term 2^{-n} in the denominator and keeping only the first two terms in the nominator, we have

$$\overline{Gain}(n,q) \approx \frac{n-q}{2(n-1)} \quad \text{(A2 3)}$$

The above approximation overestimated the gain by less than 2% for the cases of Table 3 1

References

Aho79a
Aho, A V and J D Ullman, "Optimal Partial Match Retrieval When Fields are Independently Specified," *ACM TODS*, vol 4, no 2, pp 168-179, June 1979

Bentley75a
Bentley, J L, "Multidimensional Binary Search Trees Used for Associative Searching," *CACM*, vol 18, no 9, pp 509-517, Sept 1975

Cardenas75a
Cardenas, A F, "Analysis and Performance of Inverted Data Base Structures," *CACM*, vol 18, no 5, pp 253-263, May 1975

Fagin79a
Fagin, R, J Nievergelt, N Pippenger, and H R Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files," *ACM TODS*, vol 4, no 3, pp 315-344, Sept 1979

Faloutsos85a
Faloutsos, C, "Gray Codes for Partial Match and Range Queries," *IEEE Trans on Software Engineering*, 1985 Submitted for publication

Gilbert58a
Gilbert, E N, "Gray Codes and Paths on the n-Cube," *Bell System Technical Journal*, vol 37, no 3, pp 815-826, May 1958

Gray53a
Gray, F, *Pulse Code Communications*, U S Patent 2632058, March 17, 1953

Larson78a
Larson, P, "Dynamic Hashing," *BIT*, vol 18, pp 184-201, 1978

Larson82a
Larson, P A, "Performance Analysis of Linear Hashing with Partial Expansions," *ACM TODS*, vol 7, no 4, pp 566-587, Dec 1982

Litwin80a
Litwin, W, "Linear Hashing A new Tool for File and Table Addressing," *Proc 6th International Conference on VLDB*, pp 212-223, Montreal, Oct 1980

Lloyd80a
Lloyd, J W, "Optimal Partial-Match Retrieval," *BIT*, vol 20, pp 406-413, 1980

Lloyd82a
Lloyd, J W and K Ramamohanarao, "Partial-Match Retrieval for Dynamic Files," *BIT*, vol 22, pp 150-168, 1982

Martin79a
Martin, G N N, "Spiral Storage Incrementally Augmentable Hash Addressed Storage," Theory of Computation, Report No 27, Univ of Warwick, Coventry, England, March 1979

Nievergelt84a
Nievergelt, J, H Hinterberger, and K C Sevcik, "The Grid File An Adaptable, Symmetric Multikey File Structure," *ACM TODS*, vol 9, no 1, pp 38-71, March 1984

Orenstein84a
Orenstein, J A and T H Merrett, "A Class of Data Structures for Associative Searching," *Proc of SIGACT-SIGMOD*, pp 181-190, Waterloo, Ontario, Canada, April 2-4, 1984

Ramamohanarao83a
Ramamohanarao, K, J W Lloyd, and J A Thom, "Partial-Match Retrieval Using Hashing and Descriptors," *ACM TODS*, vol 8, no 4, pp 552-576, Dec 1983

Reingold77a
Reingold, E M, J Nievergelt, and N Deo, *Combinatorial Algorithms Theory and Practice*, Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1977

Rivest76a
Rivest, R L, "Partial Match Retrieval Algorithms," *SIAM J Comput*, vol 5, no 1, pp 19-50, March 1976

Robinson81a
Robinson, J T, "The k-D-B-Tree A Search Structure for Large Multidimensional Dynamic Indexes," *Proc ACM SIGMOD*, pp 10-18, 1981

Rothnie74a
Rothnie, J B and T Lozano, "Attribute Based File Organization in a Paged Memory Environment," *CACM*, vol 17, no 2, pp 63-69, Feb 1974