A file organization scheme based on composite perfect hashing, which guarantees single access retrieval from external files has been proposed recently. The ideal retrieval performance is achieved by using an auxiliary internal table and direct perfect hashing. In this paper, we explore systematic methods of finding direct perfect hashing functions. Experimental results indicate that the proposed methods are practical.

A hashing function is said to be ‘perfect’ for a given key set and table, if it causes no overflows. Perfect hashing yields ideal retrieval performance since there are no overflows to be handled. Perfect hashing was originally defined and investigated by Sprugnoli (1977). He gave algorithms for finding two different kinds of perfect hashing functions. The other early work in this regard are reported Jaeschke (1981) and Cichelli (1980). Because of the complexity of the computations involved, all these methods considered the storage of small static sets (10 to 20 elements), such as a table of reserved words in a compiler.

The difficulty encountered in, and the complexity of finding perfect hashing functions led to theoretical investigations in this regard. Mairson (1983, 1984) proved that Ω (n) is the lower bound for the program size of searching a table with guaranteed single access retrieval. Fredman, Kolmos and Szemeredi (1982) gave a data structure which required O(n) space and enables retrievals is O(1) time.

Ramakrishna and Larson (1989) considered using perfect hashing for organizing external files. The perfect hashing schemes were classified into two categories: (a) Direct Perfect Hashing; and (b) Composite Perfect Hashing. Composite Perfect Hashing uses auxiliary storage and retrievals involve two levels of access. They proposed a file organization scheme based on composite perfect hashing. This scheme guarantees single access retrieval from external files at the cost of storage space in an internal memory and increased cost of insertions. The basic idea is to partition the large file into a number of small groups and store each group separately using direct perfect hashing. The details of individual groups are stored in internal table. The computational and I/O cost of the scheme have been shown to be competitive with other traditional hashing schemes. A trial and error method of finding direct perfect hashing functions was proposed and investigated. Investigation of systematic methods of finding direct perfect hashing functions was left as open problem. This is the main problem addressed in this paper.
Outline of the Paper

In the next section we give the QR algorithm for finding Quotient Reduction perfect hashing functions for external files. It forms a part of the Remainder Reduction method of finding direct perfect hashing functions, presented in the labeled section. Both methods are extensions of Sprugnoli’s methods to external files. Experimental results presented at the end of the section titled Remainder Reduction Method indicate that the cost of finding perfect hashing functions and the resulting storage utilization are in acceptable ranges. The dynamic behavior of the proposed methods is also investigated.

Quotient Reduction Method

Let $I = \{x_1, x_2, \ldots, x_n\}$ denote the set of keys to be stored in a hash table, $m$ the number of buckets in the table, and $b$ the bucket size. Without loss of generality, we assume that the set $I$ is in sorted order. We use $[p..q]$ to denote the interval $[p, p+1, p+2, \ldots, q-1, q]$, $(p < q)$, the length of which is $q-p+1$. The load factor $\alpha$, is the ratio $n/m$, often expressed as a percentage.

Sprugnoli defined functions of the form $h(x) = \lfloor (x+s)/N \rfloor$, $s$ and $N$ are constants, as Quotient Reduction hashing functions. He gave a method of determining $s$ and $N$ for any given key set $I$ such that the Quotient Reduction hashing function defined is perfect. His algorithm gives the optimal values of $s$ and $N$ to achieve the maximum storage utilization possible. His algorithm is restricted to the special case of $b = 1$ (and could handle 10-15 element sets only). In the following we develop algorithms which can handle key sets having a few hundred elements, when the bucket size is large. It forms a part of the Remainder Reduction method, a practical method of finding direct perfect hashing functions, discussed in the next section. We first define a set of admissible increments of the Quotient Reduction hashing function for a given $I$, $b$, and $N$ if and only if the set of admissible increments $\text{J}_i$, $1 \leq i \leq n-b$, have at least one element in common: i.e.,

\[ \bigcap_{i=1}^{n-b} \text{J}_i \neq \emptyset \]

**Proposition 1**

There exists a quotient reduction perfect hashing function for a given $I$, $b$, and $N$ if and only if the set of admissible increments $\text{J}_i$, $1 \leq i \leq n-b$, have at least one element in common: i.e.,

\[ \bigcap_{i=1}^{n-b} \text{J}_i \neq \emptyset \]

**Proof**

The “only if” part is true, since if $h(x) = \lfloor (x+s)/N \rfloor$ is a perfect hashing function for the key set, then for every $x_i, \lfloor (x_i + s)/N \rfloor < \lfloor (x_{i+b} + s)/N \rfloor$. (Since, no more than $b$ keys hash into any bucket.) Thus, $s \in \text{J}_j$ for $i = 1, 2, \ldots, n-b$, and hence $s \in \bigcap \text{J}_i$.

The “if” part can be proved as follows. Let $s$ be a common element of all $\text{J}_i$’s, $i = 1, 2, \ldots, n-b$. For every $x_i, \lfloor (x_i + s)/N \rfloor < \lfloor (x_{i+b} + s)/N \rfloor$. And hence no more than $b$ keys hash into any bucket under the hashing function $h(x) = \lfloor (x+s)/N \rfloor$. $\square$

**Proposition 2**

For a given $I$, $b$, and $N$, the sets of admissible increments $\text{J}_i$, $1 \leq i \leq n-b$, are given by

\[ \text{J}_j = \{ t_i \mid t_i = u_i - x_{i+b} \mod N \} \] \hspace{1cm} (2.1)

where $\delta_i = x_i - x_j$ and $u_i$ defined by $0 \leq u_i < \sum_{j=i}^{i+b-1} \delta_j$.

(Note: the idea of $u_i$ becomes more clear in example 1).

**Proof**

The $\text{J}_j$ defined above is equivalent to

\[ \text{J}_j = \{ t_i \mid t_i = u_i - x_{i+b} + kN \} \] \hspace{1cm} (2.2)

where $t_i$ is an integer, and let $k = \lfloor (x_{i+b} + t_i) / N \rfloor$. $t_i$ satisfies the condition $x_{i+b} + t_i \geq kN$. (2.3)

If $t_i$ satisfies the condition, $x_i + t_i < kN$, then $x_i$ and $x_{i+b}$ hash into two different buckets under the hashing function $h(x) = \lfloor (x_i + t_i) / N \rfloor$. Hence, any $t_i$ satisfying (2.3) and (2.4) is an admissible increment of $x_i$. Consider an element of the set $\text{J}_j$ as defined, $(u_i - x_{i+b} + kN)$. This element satisfies condition (2.3), since $x_{i+b} + (u_i - x_{i+b} + kN)$

Admissible increments

In order that a hashing function is perfect for the key set if the keys $x_i$ and $x_{i+b}$ should not hash into the same bucket, for $1 \leq i \leq n-b$. For given a quotient $N$, an integer $t_i$ is said to be an admissible increment of $x_i$ if

\[ \lfloor (x_i + t_i) / N \rfloor \neq \lfloor (x_{i+b} + t_i) / N \rfloor, \quad 1 \leq i \leq n-b, t_i \in \text{J}_i. \]

In other words, $t_i$ is a translation value which adjusts $x_i$ and $x_{i+b}$ into two different intervals $[(p-1)N..pN-1]$ and $[(q-1)N..qN-1]$, where $p$ and $q$ are integers, $p < q$. We use $\text{J}_i$ to denote the set of all admissible increments of $x_i$. In all, there are $n-b$ sets of admissible increments for a given $I$ and $b$. 

Consider an element of the set $\text{J}_j$ as defined, $(u_i - x_{i+b} + kN)$. This element satisfies condition (2.3), since $x_{i+b} + (u_i - x_{i+b} + kN)$
The element satisfies condition (2.4), since

\[ x_i + (u_i - x_{i+b} + kN) = (x_i - x_{i+b}) + u_i + kN < kN, \quad (\text{Since } 0 \leq u_i < \sum \delta_i). \]

Figure 1 shows the admissible increments for \( x_i \) with \( b = 2 \).

The keys can be moved to the left by \( x_{i+2} - kN \), and to the right by \( kN - x_i \) without affecting the separating boundary. The range of the admissible increments is given by

\[ x_{i+2} - kN + kN - x_i = x_{i+2} - x_i = \delta_i + \delta_{i+1}. \]

Each \( J_i \) is a subset of integers \([0..N-1]\). For each \( x_i, 1 \leq i \leq n-b \), the set of admissible increments, \( J_i \), can be computed using (2.1). If the length of the interval between \( x_{i+b} \) and \( x_i \) is equal to or greater than the quotient \( N \), \( \sum_{j=i}^{i+b} \delta_j \geq N \) (readily given by the 3rd difference).

Example 1

Consider the key set

\[ I = \{31, 58, 67, 123, 142, 146, 154, 187, 198, 220\} \], and let \( b = 3 \).

1st difference 27 9 56 19 4 8 33 11 22
2nd difference 36 65 75 23 12 41 44 33
3rd difference 92 84 79 31 45 52 66

Suppose \( N = 74 \) (later we will see how to choose the value of \( N \)). \( J_4 \) is all \([0..73]\), since the corresponding third difference are \( \geq 74 \). \( J_4 \) is computed as follows. Since \( \sum_{i=1}^{6} \delta_i = 31 \) (readily given by the 3rd difference),

we have \( J_4 = \{ t_4 \mid t_4 = u_4 - 154 \mod 74 \text{ and } 0 \leq u_4 \leq 31 \} = [0..24] \cup [68..73] \).

Similarly,

\[ J_5 = [0..5] \cup [35..73] \]
\[ J_6 = [0..1] \cup [24..73] \]
\[ J_7 = [2..67] \]

By taking the intersection of all the \( J_i \)’s, we obtain \( J_4 \cap J_5 \cap J_6 \cap J_7 = \emptyset \).

It follows from proposition 1 that a Quotient Reduction perfect hashing function does not exist for the given key set with \( N = 74 \). However, with \( N = 73 \),

\[ J_4 = \{ t_4 \mid t_4 = u_4 - 154 \mod 73 \text{ and } 0 \leq u_4 \leq 31 \} = [0..22] \cup [65..72] \]

For given \( I \) and \( b \), the minimum number of buckets \( m^* \) is given by \( m^* = \lceil n/b \rceil \).

The upper bound \( N^* \) which yields \( m^* \) buckets is given by

\[ m^* - 2 = \left\lfloor \frac{r}{N^*} + \frac{r}{N^*} \right\rfloor + 2 \quad (\text{where } r \mod 0). \] (2.6)

Therefore,

\[ m^* - 2 = \left\lfloor \frac{r}{N^*} \right\rfloor + 2 \quad (r = (m^*-2)N^* + r \mod N^*). \]

Since \( 1 \leq r \mod N \leq N-1 \),

\[ (m^*-2)N^* + 1 \leq r \leq (m^*-1)N^* + N^* - 1 \]

\[ \left\lfloor \frac{r+1}{m^*-1} \right\rfloor \leq N^* \leq \frac{r-1}{m^*-2}. \]

Considering \( N^* \) takes integer values only, the upper bound is given by

\[ N^* = \left\lfloor \frac{r-1}{m^*-2} \right\rfloor = \left\lfloor \frac{r-1}{n/b - 2} \right\rfloor. \] (2.7)

Solution Space of the Quotient Reduction Method

Figure 2 illustrates the relationship between quotient \( N \), admissible increment \( s \), and the number of buckets \( m \).
The horizontal axis represents the value of \((x_1 + s) \mod N\), with \(s\) as a variable. The vertical axis represents the value of the quotient \(N\). The first row of the figure corresponds to \(N = N^*\). The left hand side of the boundary (hatched portion) in the first row corresponds to the space in which the following inequality holds,
\[ (x_1 + s) \mod N^* \leq (N^* - 1) - r \mod N^*. \]
The values \(N^*\) and \(s\) in this space yields \(m^* - 1\) buckets. The space in which the following inequality
\[ (x_1 + s) \mod N^* > (N^* - 1) - r \mod N^* \]
holds is in the right hand side of the boundary. The values \(s\) and \(N (= N^*)\) in this space yields \(m^*\) buckets. Similarly, each row represents the solution space for a different \(N\).

**Algorithm QR (Quotient Reduction)**

We are now ready to introduce our algorithm QR to generate a perfect hashing function for the given set of keys \(I\). The basic idea of this algorithm is to search the solution space of \(N\) and \(s\) to obtain a perfect hashing function starting with \(m^*\) buckets, and proceeding with \(m^*+1\) buckets, etc.

**Algorithm QR**

**STEP 1** \{ Initialization \}
Compute the range of the keys: \(r := x_n - x_1\).
Compute the minimum number of buckets: \(m := \lceil n/b \rceil\)

**STEP 2**
Compute the upper bounds and lower bounds of \(N\) corresponding to \(m\) buckets.
\[ NU_2 := \left\lfloor \frac{r}{(m-2)} \right\rfloor \]
\[ NL_2 := \left\lfloor \frac{(r+1)/(m-1)}{m} \right\rfloor \]
\[ NU_1 := \left\lfloor \frac{r}{(m-1)} \right\rfloor \]
\[ NL_1 := \left\lceil \frac{(r+1)/m}{m} \right\rceil \]
for \(N := NL_1\) to \(NU_1\) do
Compute the set of the admissible increments \(J_i\) (as described in proposition 2) with \(N\).
Form a subset \(J'\), \(J' = \{ t \mid r \mod N + (x_1 + t) \mod N > N - 1 \} \). (This is done to ensure that we consider only those admissible increments which give the minimum value of \(m\).)
Take the intersection, \(JI = n-b \cap J_i \cap J'1\)
if not empty then goto STEP 3
end for

for \(N := NL_2\) to \(NU_2\) do
Compute the set of the admissible increments \(J_i\) (as described in proposition 2) with \(N\). Form a subset \(J'\), \(J' = \{ t \mid r \mod N + (x_1 + t) \mod N \leq N - 1 \} \).
\{ This is done to ensure that we consider only those admissible increments which give the value of \(m\). \}
Take the intersection, \(JI = n-b \cap J_i \cap J'1\)
if not empty then goto STEP 3
end for

\(m := m + 1\)
goto STEP 2

**STEP 3**
\(s := \) Choose a value from intersection set. (See description below)
The perfect hashing function is \( h(x) = \left\lfloor \frac{(x+s)/N}{N} \right\rfloor \).

A few clarifications about the algorithm follow. The range of \( N \) that corresponds to \( m \) buckets is determined as follows.

When the condition \( \{ r \mod N + (x_1 + J) \mod N \} \geq N \) holds, the range of \( N \) is from \( NU_1 \) to \( NL_1 \).

When the condition \( \{ r \mod N + (x_1 + J) \mod N \} \leq N-1 \) holds, the range of \( N \) is from \( NU_2 \) to \( NL_2 \).

If the intersection is not empty, any element of the set \( J \) is acceptable (all of them give the same load factor). However, to obtain a more uniform distribution of the keys, choose \( s^* \in J \), which makes the interval covered by the first bucket, \( [x_1 .. p(N-1)] \), as close as possible with that of the last bucket, \( [qN .. x_n] \), where \( p = h(x_1) + 1 \) and \( q = h(x_n) \).

The aim of this is to minimize the probability of rehashing (rehashing is dealt with in greater detail in section 4). This can be accomplished by choosing \( s^* \), so that
\[
s^* = \min_{j \in J} \{ N - (x_1 + j) \mod N - (x_n + j) \mod N \}.
\]

In order that \( x_1 \) hashes into bucket address 0, we need the final transformation \( s = s^* - N \{ (x_1 + s^*) \div N \} \).

Example 2

We illustrate the above procedure using a small set (note that in practice a much larger set will be handled). Consider the key set \( I = \{31, 58, 67, 123, 142, 146, 154, 187, 198, 220\} \) with \( b = 3 \).

We have \( n = 10 \) and the range of keys, \( r = x_{10} - x_1 \)
\( = 189 \).

The minimum number of buckets is \( m^* = \left\lceil \frac{n}{b} \right\rceil = 4 \).

The upper bound \( NU_1 := \left\lceil \frac{r}{(m-2)} \right\rceil = \left\lceil \frac{189}{(4-2)} \right\rceil = 94 \).
The lower bound \( NL_1 := \left\lfloor \frac{r}{(m-1)} \right\rfloor = \left\lfloor \frac{189}{(4-1)} \right\rfloor = 63 \).

The upper bound \( NU_2 := \left\lceil \frac{r}{(m-1)} \right\rceil = \left\lceil \frac{189}{(4-1)} \right\rceil = 64 \).
The lower bound \( NL_2 := \left\lfloor \frac{r}{m} \right\rfloor = \left\lfloor \frac{189}{4} \right\rfloor = 48 \).

The solution space is shown in Figure 3. The search starts with the value \( N = NL_2 = 48 \), and the set of admissible increments \( J = \{17, 18, 19\} \) is found with \( N = 48 \) in STEP 2. We move to STEP 3, and the element [18] is chosen which is transformed as \( s = 18 + 48 \{ 31 + 18 \div 48 \} = 30 \), so that \( h(x_1) = 0 \). We thus have the Quotient Table 1:

<table>
<thead>
<tr>
<th>bucket #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>31</td>
<td>58</td>
<td>67</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>142</td>
<td>146</td>
<td>154</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>198</td>
<td>220</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Hash Table for the example 2

Figure 3 Solution for Example 2

Reduction perfect hashing function,
\( h(x_1) = \left\lfloor \frac{(x_1 - 30)/49}{N} \right\rfloor \)

The hash table for this example is shown in Table 1.

A heuristic for efficient QR Algorithm (EQR)

We see from (2.7) that the upper bound \( N^* \) of the quotient \( N \) is proportional to the range \( r \) of keys. Since, every \( N \) is examined sequentially in the algorithm QR, the worst case complexity makes the algorithm impractical, although the average case is much better. We repeat that the QR algorithm as a whole is impractical in general, but its importance is that it is a part of the next algorithm which is practical. We experimented with several heuristics and the following heuristic resulted in a considerable speed up of the algorithm.

The algorithm starts with \( m^* + 1 \) buckets and proceeds with \( m^* + 2 \), and so on. For each value of \( m \), only one value of \( N, \ N = \left\lfloor \frac{r}{(m-1)} \right\rfloor \) is considered. (It can be readily verified that this value of \( N \) corresponds to the largest range of the possible admissible increments). An attempt is made to find an admissible increment. If the search fails, we proceed with the next value of \( m \). If the search succeeds with \( m_i \) and \( N = N_i \), we proceed with \( m_{i-1} \). (Obviously, this is an attempt at increasing the storage utilization). For this special case, we examine a range of \( N_i \) from \( N = N_i + 1 \) to \( \left\lfloor \frac{r}{(m_i - 2)} \right\rfloor \). Detailed algorithm is given by Bannai (1988), and we do not think its inclusion would further enlighten the ideas presented here. The following example illustrates the EQR algorithm.
Example 3
Consider the key set $I = \{31, 58, 67, 123, 142, \ldots\}$ with $b = 3$. We have $n = 10$ and the range of keys, $r = x_{10} - x_1 = 189$. (The same data set as in example 2). The minimum number of buckets is $m^* = \lceil n/b \rceil = 4$. First, we consider $m = m^* + 1$ buckets. The corresponding $N$ is $N = \lceil r/(m-1) \rceil = \lceil 189/(5-1) \rceil = 47$. With this value of $N$, the search for an admissible increment succeeded, and we obtained the perfect hashing function which requires five buckets. The algorithm proceeds with $m = m_{\ast} + 1$ buckets, and the values of $N$ ranging from 48 ($= N_{\ast} + 1$) to 63 ($= \lceil r/(m-1) \rceil$). A set $J = \{17,18,19\}$ was determined to be the set of admissible increments with $N = 48$. The value $[18]$ was chosen satisfying

$$s^* = \min \{N - (31 + j) \mod 48 - (220 + j) \mod 48\}$$

and transformed to $s = 18 - 48\{ (31 + 18) \div 48 \} = -30$. Consequently, the perfect hashing function is obtained as $H(x) = \lceil (x - 30)/48 \rceil$. (This hashing function is the same as that in example 2.)

### Table 2: Probability of having no collisions for given $M$ and $n$

<table>
<thead>
<tr>
<th>$M$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.53</td>
</tr>
<tr>
<td>250</td>
<td>0.99</td>
</tr>
<tr>
<td>100</td>
<td>0.99</td>
</tr>
</tbody>
</table>

The probabilities of having no collisions for $n=500$, 250, and 100 are shown in Table 2. For example, $M$ should be greater than $2^{21}$ when $n = 500$ in order that probability of collisions is small. We do not gain the advantages of the Remainder Reduction method with this large a value of $M$. However, we can tolerate some collisions by the transformation $(3.2)$. If the maximum number of identical keys is less than or equal to the bucket size $b$, then in general there exists a quotient reduction perfect hashing function. On the other hand, the probability that more than $b$ keys collide under this transformation is almost zero. Let $P(\alpha , m, b)$ denote the probability that none of the urns contain more than $b$ balls, when $n, n=\alpha bm$, balls are tossed randomly into $m$ urns. The following approximate formula is derived by Ramakrishna (1987),

$$P(\alpha , m, b) = e^{-n Pov(\alpha,b)}$$

where $Pov(\alpha,b) = (b\alpha)^{b+1}/(b+1)! \cdot e^{-b\alpha} \{b+2/(b(1-\alpha)+2\}$. (3.4)

For example, when $n=500$, $m=500$, and $b=10$, The probability that more than $b$ keys collide is given by

$$1-P(0.1, 500, 10) = 1 - e^{-500Pov(0.1, 10)}$$

= $1 - e^{-5.03\times 10^{-6}}$ = a negligibly small quantity

Now we are ready to introduce our Remainder Reduction algorithm. To implement this method, it is necessary to apply the transformation $x' = (qx_{i}) \mod M$ for each $x_{i}$, $1 \leq i \leq n$, and sort the keys before applying the Quotient Reduction method. Let $\{w_1, w_2, \ldots, w_n\}$ be the sorted set of keys. The constants $q$ and $M$ are chosen to be prime numbers.

### Algorithm RR

**STEP 1**
Choose $M$ to be a prime number larger than $n$. 

Complexity of the Algorithms QR and EQR

Each intersection operation requires $O(n^2)$ steps. In the worst case, the loop on $N$ is bounded by the upper bound of $N_{\ast} = \lceil r/(n/b -2) \rceil$. Thus, the worst case complexity of the QR algorithm is $O(rbn)$. The complexity of the EQR algorithm is also the same essentially, but for the constant of proportionality. These algorithms are used in the Remainder Reduction method discussed in the next section. The value of $r$ is controlled and hence the complexity. The space requirement of these algorithms is $O(n)$. 

### Remainder Reduction Method

The Remainder Reduction method overcomes the problems of low storage utilization and large time complexity of the Quotient Reduction methods: The basic idea of this method is to scramble the set of keys to obtain a more uniform distribution within a narrow range, followed by Quotient Reduction perfect hashing. Since we can control the reduced key size, the complexity can be controlled. Remainder Reduction perfect hashing follows.

$$h(x_{i}) = \lceil (qx_{i}) \mod M + s \rceil /N \rceil$$

where $q$, $M$, $s$, $N$ are constants to be determined. The transformation $x' = (qx_{i}) \mod M$ (3.2) accomplishes the scrambling mentioned above by choosing a $q$ and $M$ appropriately (more details in this regard are given by Ramakrishna (1988)). The transformation may cause collisions, i.e., more than two distinct primary keys are transformed into the same numbers. The probability $p$ of no collision occurring by the transformation can be obtained as follows.

$$p = M^p / M^n = M(M-1)(M-2)\ldots(M-n+1)/M^n$$

For small positive $x$, we have $log (1-x) = -x$, $log_p = 1+2+\ldots+(n-1)/M = n(n-1)/2M$. We obtain

$$p = e^{n(n-1)/2M}.$$ (3.3)

Table 2: Probability of having no collisions for given $M$ and $n.$

<table>
<thead>
<tr>
<th>$M$</th>
<th>$p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.53</td>
</tr>
<tr>
<td>250</td>
<td>0.99</td>
</tr>
<tr>
<td>100</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Choosing \( M \) around \( n \) has proven adequate in practice (which can be readily verified by substituting in (3.4)), \( q \) is chosen to be a prime \( \leq M \).

**STEP 2**

Transform keys  
\( \{ x_1, x_2,\ldots, x_n \} \) into  
\( \{ x'_1, x'_2,\ldots, x'_n \} \) using  
\( x' = (q x_i \mod M) \).

Sort the keys  
\( \{ x'_1, x'_2,\ldots, x'_n \} \) in non-decreasing order,  
to obtain  
\( \{ w_1, w_2,\ldots, w_n \} \).

**STEP 3**

Apply the algorithm QR to the sorted keys  
\( \text{QR}(n, \{ w_1, w_2,\ldots, w_n \}) \)

Perfect hashing function  
\( h(x_i) = \lfloor (q x_i \mod M + s) / N \rfloor \)

In the extremely unlikely event of the choice of \( q \) and \( M \) resulting in a large number of collisions, a different \( q \) is chosen and the algorithm is repeated. The complexity of the Remainder Reduction Algorithm is \( O(nM) \), and the space requirement is \( O(n) \).

**Experimental Results**

To test our algorithms, we conducted several experiments using the following real life files.

(A) Userids from IBM/CMS System at the University of Waterloo (12,000 keys)

(B) Userids from the msudoc at MSU (600 keys)

All the data was in alphanumeric form and the length of individual keys varied from 2 to 12 characters. The keys were converted into 2 byte long unique integers by the RADIX Convert method. The keys in each file were divided into \( g \) groups using hashing functions of the form  
\( H(x) = ((c x + d) \mod p) \mod g \),  
where \( p \) is a large prime number. (The constants used were  
\( c = 314559, d = 27182, p = 65521, g = 9 \)) (This was done to simulate the partitioning in composite external perfect hashing scheme). For each of the nine groups obtained, Remainder Reduction direct perfect hashing functions were generated (The RR algorithm used both QR and EQR). Several values of \( q \) and \( M \) were used in this experiment, and we observed that the value of \( q \) and \( M \) had little effect on the result. The average load factor obtained for the nine groups of file (A) are shown in Table 3 along with the values of the constants. The third row gives the ratio of the average load factor obtained by two sets of results.

Perfect hashing functions were generated for the keys in file (B) without any partitioning (since the file size is small). Table 4 gives the load factors achieved along with the constants used.

We observe that load factors above 70% can be achieved almost always, and with larger bucket sizes, the load factor above 80% mark can be obtained. The RR with EQR algorithm gives the same load factor as that of RR with QR, but it runs 1 to 4 times faster. It may be observed that the parameter values correspond to those considered and analyzed in the external perfect hashing scheme presented by Ramakrishna and Larson (1988). We thus conclude that the proposed algorithms are practical for finding direct perfect hashing functions and hence may be used in file organization based on composite perfect hashing.

**Implementation Considerations**

**Rehashing during insertions**

When a new key has to be inserted into the file, the group to which it belongs is first determined, and the key is hashed using the perfect hashing function of the group. If the key hashes into a non-full bucket, then the old perfect hashing function continues to be perfect for the enlarged key set. If the key hashes into a full bucket, or its address is beyond the range of the hash table, a new perfect hashing function has to be found for the enlarged key set. This is called as rehashing Ramakrishna and Larson (1989) and further details may be found there. Under the Remainder Reduction perfect hashing functions, the probability \( P_R \) of an insertion causing rehashing is given by  
\[
P_R = \frac{R_o + R_r}{M}, \quad (4.1)
\]

where \( R_o \) is the range in \([0, M-1]\) not covered by the keys stored in the table, and \( R_r \) is the range covered by the keys stored in the full buckets (of course, this is under the usual assumption that the key to be inserted in equally likely to be mapped to any integer in the range \([0, M-1]\) under the initial transformation. It follows that,  
\[
R_o = \max(s,0) + \max((M-1) - (m+1)N, 0) \quad (4.2)
\]

and  
\[
R_r = \sum_{i \in F} R_{ri} \quad (4.3)
\]

where \( R_{ri} \) is the range covered by bucket \( i \), and \( F \) is the set of full buckets, and is given by  
\[
R_{ri} = N + \min(s,0) \quad \text{(the first bucket)}
\]
\[
N + \min(M-1) - (m+1)N, 0) \quad \text{(the last bucket)}
\]
\[
N \quad \text{(other buckets)}
\]

It should be noted that if \( m \) is large, the probability \( P_R \) approaches the ratio \((\text{number of full buckets}) / m\). Table 5 lists the probabilities \( P_R \) computed using (4.1). The data was obtained from the experiments described in the last section.
Table 3: Average load factor of nine groups of file (A) (percentage)

Table 4: Load factor for file (B) (percentage)
We observe that the probability of rehashing is quite small on the whole, and is smaller for larger key sets. With $b=50$, only one in about 15 insertions causes a rehash when the group size is 500 records. These results are comparable to those obtained for the trial and error method by Ramakrishna and Larson (1989) and hence their conclusion about the low amortized cost per insertion is applicable in our case also.

**Simulation of a file loading**

In order to study the practical performance of file organization using composite perfect hashing under RR direct perfect hashing, we implemented the hashing schemes and simulated file loading with one insertion at a time. This also enabled to study the dynamic behavior of the proposed direct perfect hashing methods. The keys of file (A) were used for experiments and header table size was fixed at 9 (i.e., 9 groups in total). The keys were inserted one at a time and if necessary the group was rehashed. Thus, the average group size increases gradually starting from zero. The overall load factor of the file was noted after each insertion. Figure 4 plots the overall load factor of the file as a function of the average group size.

The oscillations in the plots are due to fragmentation. Peaks and valleys appear when the number of records in the group is around $b$, $2b$, $3b$, ... The oscillations subside with increasing number of records. These plots show no surprises and confirm our results, in the previous sections, of the proposed direct perfect hashing methods. Load factors above 80% can be achieved for bucket sizes greater than 20. About one in 10 to 20 insertions necessitate rehashing, resulting in low amortized cost per insertion. We conclude that the use of the proposed direct perfect hashing methods are practical.

**Conclusions**

We presented two systematic methods of finding direct perfect hashing functions for external files. The Remainder Reduction method of finding direct perfect hashing function yields good load factors and the computational costs are in practically acceptable ranges. In file organizations using composite perfect hashing, the Remainder Reduction perfect hashing may be used for those groups which present worst case situations to the trial and error method.

**References**


perfect hashing functions. Comm. of the ACM, 24(12), 829-833


M.V. Ramakrishna is an Assistant Professor in the Computer Science Department at the Michigan State University. Between 1976 and 1981, he worked as an Engineer in the Computer Aided Design group of the Bharat Electronics Limited, Bangalore, India. His current research interests include file structures, hashing techniques, text searching and databases in general. He has published in many journals, such as ACM Trans. on Database Systems, Commn. of the ACM, IEEE Transactions on Knowledge and Data Engineering, Information Processing Letters, and the Commn. in Statistics.

Yuchi Bannai is a Senior Engineer in the Vision and Image Processing Division, Canon Incorporated., Japan.
Related Content

A Template for Defining Enterprise Modelling Constructs
[www.irma-international.org/article/template-defining-enterprise-modelling-constructs/3310/](www.irma-international.org/article/template-defining-enterprise-modelling-constructs/3310/)

A Systematic Approach to Flexible Specifications, Composition, and Restructuring of Workflow Activities
[www.irma-international.org/article/systematic-approach-flexible-specifications-composition/3304/](www.irma-international.org/article/systematic-approach-flexible-specifications-composition/3304/)

Managing Temporal Data
[www.irma-international.org/chapter/managing-temporal-data/20685/](www.irma-international.org/chapter/managing-temporal-data/20685/)

INDUSTRY AND PRACTICE: Why Not SISP Too?
[www.irma-international.org/article/industry-practice-not-sisp-too/51167/](www.irma-international.org/article/industry-practice-not-sisp-too/51167/)

A Measurement Ontology Generalizable for Emerging Domain Applications on the Semantic Web
[www.irma-international.org/chapter/measurement-ontology-generalizable-emerging-domain/4290/](www.irma-international.org/chapter/measurement-ontology-generalizable-emerging-domain/4290/)