Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

A Linear Hashing Enabling Efficient Retrieval for Range Queries

Ken Higuchi
Graduate School of Engineering,
University of Fukui,
Fukui-shi, Japan,
higuchi@u-fukui.ac.jp

Tatsuo Tsuji
Graduate School of Engineering,
University of Fukui,
Fukui-shi, Japan,
tsuji@pear.fuis.fukui-fu.ac.jp

***Abstract*—For efficient retrieval of data, design of the data structuring is important. Tree structures and hash tables are popular data structures. A hash table is a simple data structure and it can be retrieved very fast for an exact match query. But for a range query, the hashing scheme is necessary to search much more data blocks than other data structures. In this paper, in order to overcome this problem, the order-preserving linear hashing scheme is proposed. This hashing scheme is based on linear hashing which uses specific hash function enabling efficient retrieval for range queries. By comparing the proposed hashing scheme with the traditional linear hashing scheme, our scheme proves to provide better retrieval time for range queries.**

***Keywords*—linear hashing, dynamic hashing, range query**

I. INTRODUCTION

Nowadays, many computer users have a large amount of data, and they want to retrieve the desired target data efficiently from these data. But if the amount of the target data becomes large, the retrieval time would much increase. For efficient retrieval of data, design of the data structuring is important. Tree structures and hash tables are popular data structures. A tree structure such as a B-tree is complexly structured, but a range query against it can be performed with good response time. A hash table is simply structured and an exact match query can be performed with good response time. But for a range query, the response time is not so good. Since a lot of hashing systems employ a modulo function, the hash values of the resulted data for a range query are not the same value, so many data blocks should be accessed. Another disadvantage of hash tables is the conflict of the data whose hash values are the same. To reduce this conflict, some dynamic hashing schemes were proposed[1][2][3]. Linear hashing[3] is one of such dynamic hashing schemes. On the linear hashing, for insertions of new data records, a hash table slot can be dynamically expanded one by one. Since the whole reconstruction of the hash table is not necessary, the expanding cost of the hash table is not so high. But the disadvantage for range queries still remains.

In this paper, we propose a new linear hashing scheme called *order-preserving linear hashing*. In our hashing scheme, data is stored in numerical order. The scheme is based on the linear hashing and use specific hash function enabling efficient retrieval for range queries. By comparing the proposed hashing scheme with the traditional linear hashing scheme, our scheme proves to provide better retrieval performance for range queries.

II. LINEAR HASHING

Linear hashing[3] is a kind of dynamic hashing. It consists of a hash function ($h()$), a bucket array ($a[]$), data buckets, a fixed fraction (p), and meta-information. The data bucket consists of data blocks, and meta-information include the hash level (i), the number of data buckets (n), the number of records (r), and p is the least upper bound the average number of data in a single data bucket; i.e., r/n. Data are stored into the data block in the data bucket, the bucket array keeps the addresses of the data buckets, and its size is n. In general, the hash function $h()$ is modulo by 2^i; i.e. the hash value is the i bits suffix of the bit pattern of the input data.

In the process of data retrieval, the hash value of the target data (v) is calculated by $h()$, $a[v]$ keeps the address of the data bucket storing the target data. When $v > n-1$, $a[v]$ does not exist. In this case, $v - 2^{i-1}$ is used for the hash value. Then this data bucket is searched for retrieval. Figure 1 is an example of the linear hashing. There are three data {00000, 01011, 11010}, three data buckets, and the size of the bucket array is 3. The numbers in bucket array are hash values that specify the corresponding data buckets and they are not addresses of data buckets in this figure. In this example, since a data block can store only a single data, if two data should be stored in same data bucket, an overflow block is allocated (Figure 2.)

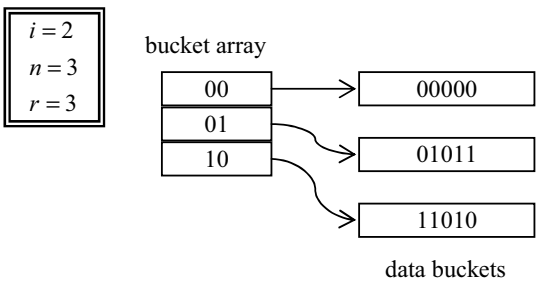

Figure 1. Liner Hashing

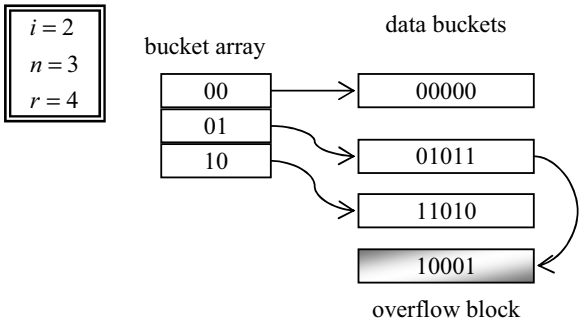

Figure 2. Liner Hashing (using overflow block)

By using overflow blocks, the conflict problem is resolved, but the retrieval time becomes worse, because it is necessary to search through the list of overflow blocks in the data bucket. In this example, to access the data "10001", the system needs to access three data objects, namely the bucket array, the first block storing "01011", and the overflow block. To reduce such excessive accesses, the linear hashing can expand dynamically the bucket array. When $p < r/n$, the bucket array is expanded to have a reference to a new data bucket. The hash value of the new data bucket is n and n is greater than other hash values. Here, let $p = 1$. In Figure 1, when a user inserts the data "10001", r becomes 4 and $p = 1 < r/n = 4/3$. Since $p < r/n$, the bucket array should be expanded. In the linear hashing scheme, the bucket array is expanded one by one, the data bucket "01" is divided into the data bucket "01" and the new data bucket "11", and n is incremented to 4. After the expansion of the bucket array, the new data "10001" is inserted to the data bucket "01" (Figure 3.)

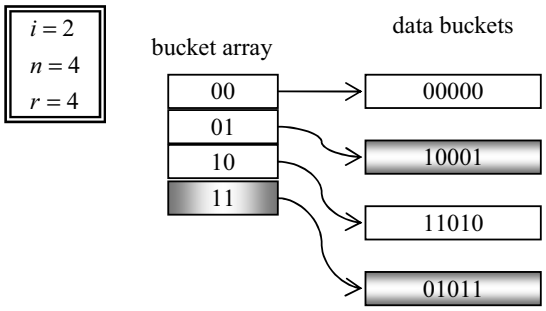

Figure 3. Bucket array expansion in Liner Hashing

When $r = 2^i$, the linear hashing cannot expand the bucket array. In this situation, to expand the bucket array, i is incremented (See Figure 4.) The hash function should be changed and all of the hash values should be recalculated according to this new hash function. In this example, "0" is added to any hash values as the prefix; e.g., "00" is changed to "000", "01" is changed to "001". After this change, a new element is added to the bucket array. In this example of Figure 4, "100" is the added new element. Since the target of the divided data bucket is only one ("01" in Figure 3, "00" in Figure 4) and other data buckets are not necessary to be modified, and the bucket array is expanded one by one, the cost of the bucket array expansion is very small.

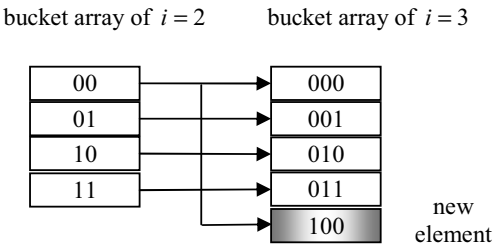

Figure 4. Expanding bucket array

But for range queries, the linear hashing has a disadvantage. If a range query [between "00010" and "00100"] is requested in the situation of Figure 3, the system has to search the three data buckets "10", "11", and "00". Furthermore, if a range query [between "00100" and "00111"] is requested, the system has to search all data buckets. In general, if the range of the query is wider than 2^i, the system has to search all data buckets, because the hash function is the modulo function by 2^i and such range of the query includes all hash values. Then the linear hashing scheme and other hashing schemes which use the modulo function share such disadvantage of the necessity of searching widely for range queries. We will call the linear hashing using the modulo function as the *traditional linear hashing.*

In the following, a range query is simplified and denoted as $rq(N, x)$. Here, N is the bit length of the retrieval data, x is the bit string which is the common prefix between the greatest lower bound and the least upper bound of the range query, and its length ($|x|$) is not more than N; i.e. $|x| \leq N$. $rq(N, x)$ is the range query which requires to search the retrieval data between $x00\cdots0$ and $x11\cdots1$, where $x00\cdots0$ and $x11\cdots1$ are the bit string whose prefix is x and whose length is N.

If $|x| = N$, $rq(N, x)$ is the exact match query and the traditional linear hashing searches only a single data bucket. In the other cases, the time complexity of the traditional linear hashing is depending on the overlap bits between x and the hash value, because x is the prefix of the bit string of the retrieval data while the hash value is the suffix of it. If x reaches to the hash value's bit (i.e. $i + |x| > N$,) then data buckets to be searched out for $rq(N, x)$ are restricted by the

overlap bits between x and the hash value, and the traditional linear hashing has to search $2^{N-|x|}$ data buckets. If $i+|x| \le N$, then there is no overlap bits between x and the hash value, and the traditional linear hashing has to search all data buckets. Hence, in many cases, the traditional linear hashing needs to search many data buckets for a range query.

III. ORDER-PRESERVING LINEAR HASHING

To overcome the above mentioned problem inherent in the traditional linear hashing, we consider a hash function other than the modulo function. We employ the division function as the hash function. By this division function, the hash value becomes a prefix of the bit string of the retrieval data. See Figure 5. By using the division function, the neighboring data in bit pattern can be stored in the same data bucket in most cases. So, the number of data buckets that should be searched for a range query can be reduced.

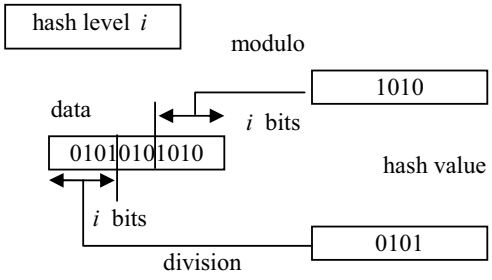

Figure 5. Hash Functions (modulo and division)

This modification of the hash function makes the retrieval process to be modified. For the retrieval data, the hash value (v) is calculated by the division function, and $a[v]$ keeps the address of the data bucket storing this data. When $v > n-1$, $a[v]$ does not exist. In this case, $v-1$ is used for the hash value. Then the corresponding data bucket is searched. Estimate the number of data buckets which are searched for $rq(N,x)$. If $i \ge |x|$, this linear hashing is sufficient to search only a single data bucket. If $i < |x|$, $2^{|x|-i}$ data buckets should be searched out. In many cases, this modified linear hashing is sufficient to search only a single data bucket for a range query.

But this modified linear hashing using the division function becomes to have a disadvantage when the bucket array is expanded. Since in the traditional liner hashing, the hash value of new data bucket is greater than other hash values, the existing bucket array doesn't have to be modified. But in the linear hashing using the division function, when the bucket array is expanded, the existing bucket array has to be modified. Figure 6 shows this situation. When i is incremented from 2 to 3, each hash value is added 0 as the suffix; e.g., "00" is changed to "000", "01" is change to "010". After this increment, a new element is added to the bucket array. In this example, "001" is new element. In this situation, the bucket array should be reconstructed; i.e., "010", "100", and, "110" have to be moved when "001" is added. If the bucket array is large, this operation is very costly, because the bucket array is the static array in general. Furthermore, this costly operation occurs whenever a new element is added not as the last one (in this case, "111".) Hence, the bucket array modification cost of this linear hashing using the division function is hither than the traditional linear hashing.

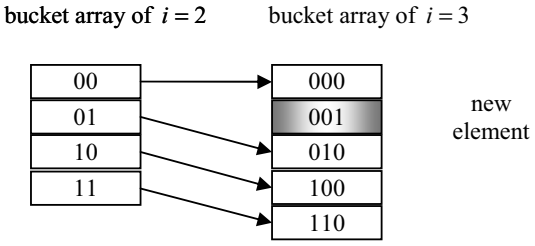

Figure 6. Expanding bucket array in the linear hashing using division function

To overcome this problem, we use the *division and bit reversal* function as the hash function. This function rd() outputs the reverse ordered bit string of the result of the division function (Figure .7)

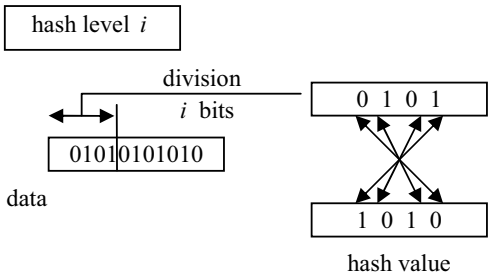

Figure 7. Hash Function: rd()

We call the linear hashing using rd() as the *order-preserving linear hashing*. Figure 8 shows the example of the order-preserving linear hashing, which stores same data in Figure 3.

On the order-preserving linear hashing, the expanding method for the bucket array is the same as the traditional linear hashing. When i is incremented, the old hash values are added "0" as the prefix and the hash value of new element of the bucket array is greater than other hash values. Then the order-preserving linear hashing inherits the advantage of the traditional linear hashing with good response time for the range query.

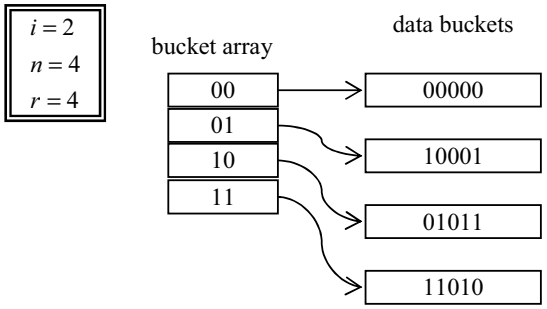

Figure 8. The Order-Preserving Linear Hashing

IV. Related Work

For efficient retrieval for range queries, many algorithms were proposed[4][5][6][7][8][9]. Division and bit reversal function is used in some schemes[4][5][6][7]. The interpolation-based order preserving hashing[4][5][6] is using the division and bit reversal function as the hash function. But the domain of the hash function is the real number between 0 and 1 in the binary bit string. While our technique can be adapted to any bit strings, and can treat any type of domains, such as character strings, integers, and floating point numbers. The extendible hash for multi-dimension data[7] is based on the extendible hashing[2]. So, it has the disadvantage that the hash table should expand to the double sized one at expansion. While our scheme is based on the linier hashing, and the hash table is added only one element at expansion. There are some schemes using the division function for Peer-2-Peer system[8][9]. They are based on DHT (Distributed Hash Table) system and use the range of data for its partition. But they don't consider the global shared hash table. Therefore they use only the division function as the hash function and are not necessary to reconstruct of the hash table. These schemes work like the prefix hash tree.

V. Time Complexity

To compare the proposed order-preserving linear hashing with the traditional linear hashing, we estimate the time complexity for the range query in the situations described in Table 1. In the estimation, we only focus on the number of data buckets that are searched in the operation for a range query, because the data buckets are stored in the secondary storage and the time complexity depends on the number of accesses to the secondary storage. For simplicity, for each hash level, r is assumed to be 2^i. Hence, the time complexity depends on $|x|$ and not on the value of x.

The time complexity of the traditional liner hashing is already estimated in Sec. II. For the range query, the time complexity of the order-preserving linear hashing is the same as the linear hashing using the division function. The timing of the bucket array expansion in the order-preserving linear hashing scheme is the same as that in the traditional linear hashing scheme and the size of the bucket array of the order-preserving liner hashing is the same as that of the traditional linear hashing too. Then we can compare the time complexity of them directory. Figure 9, 10, 11 show the time complexity of the order-preserving linear hashing and the traditional linear hashing. In these figures, OPL is the order-preserving linear hashing and TL is the traditional linear hashing. The numbers of data buckets are 2^{16} (hash level = 16), 2^{32} (hash level = 32), 2^{48} (hash level = 48).

Table 1. Parameters of linear hashing

Parameter	Value
Bit length of data	64
Hash level	16, 32, 48
$\|x\|$ of rq$(64, x)$	0,1,2,….,64

From these figures, the time complexity of OPL is less than that of TL in most cases and not greater than that of TL. In cases where $|x| = 0$ or $|x| = 64$, they are equal. Note that the curve of OPL is the point reflection to that of TL. Therefore, if the hash level is lower, the difference between the time complexity of OPL and that of TL is greater. Consider the situation that $p = 80$. In this situation, the linear hashing whose hash level is 16 can store 5,242,880 data, and the linear hashing whose hash level is 32 can store 343,597,383,680 data. Then, in many cases, the hash level doesn't reach to 32. Hence, in the real system, the order-preserving linear hashing is faster than the traditional linear hashing.

VI. Comparing with B+tree

The ordering of the data in the order-preserving linear hashing is similar to that in B+tree. But the access method to the data block of the order-preserving linear hashing is different from that of the B+tree. The B+tree can access to the data block storing the target data directly after the traversal on the index nodes. While the order-preserving linear hashing can access to the data bucket storing the target data directly after the access to the hash table then it needs to search the data bucket. But if there are many overflow blocks in the data bucket, the order-preserving linear hashing needs to search many data blocks. So, the worst case time complexity of the order-preserving linear hashing for both the exact match query and the range query is worse than that of B+tree. Furthermore, if there are many overflow blocks in the data bucket, conflicts between queries and modifications on the same data bucket occur frequently. It causes the deterioration of the performance.

Although by employing parallelism, this situation can be improved. For a range query, data blocks storing the target data in the sequence set of the B+tree should be traversed. But the order-preserving linear hashing can directly access to the data buckets storing the target data. Then the order-preserving linear hashing can process the range query in parallel, while in the B+tree, the related data blocks in the sequence set should be accessed sequentially. Hence, the throughput of the order-preserving linear hashing may be better than that of the B+tree in some cases. Fortunately, we can use distributed versions of

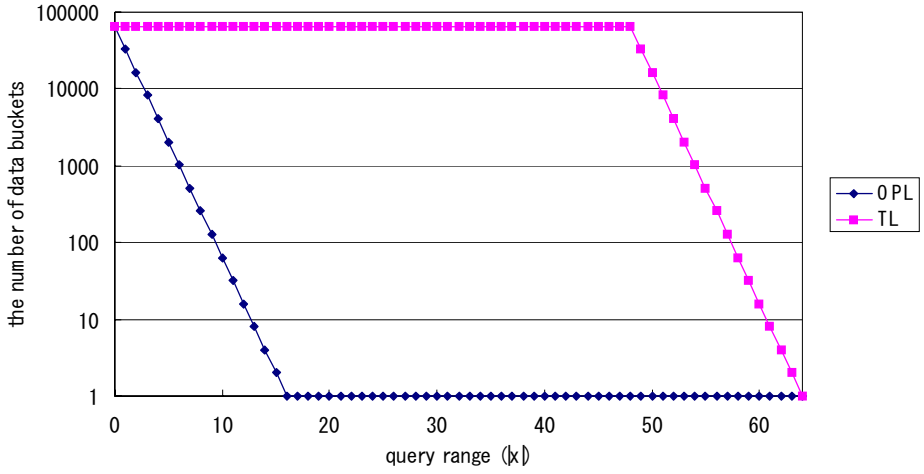

Figure 9. The Time Complexity (hash level = 16)

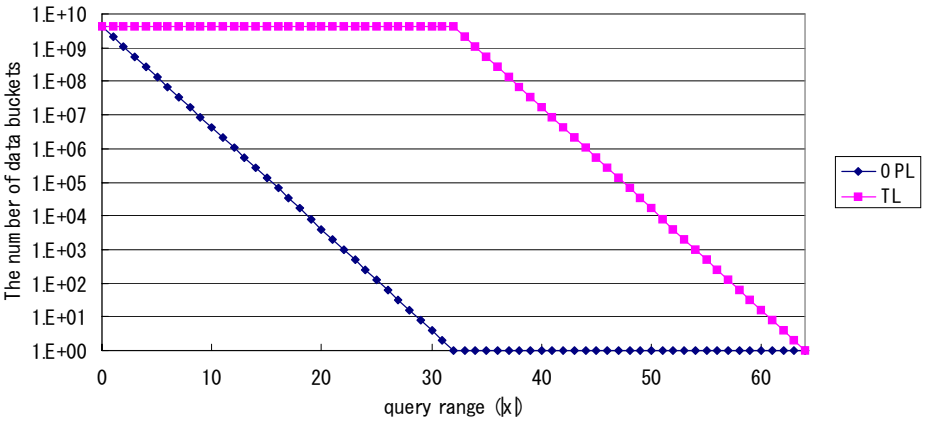

Figure 10. The Time Complexity (hash level = 32)

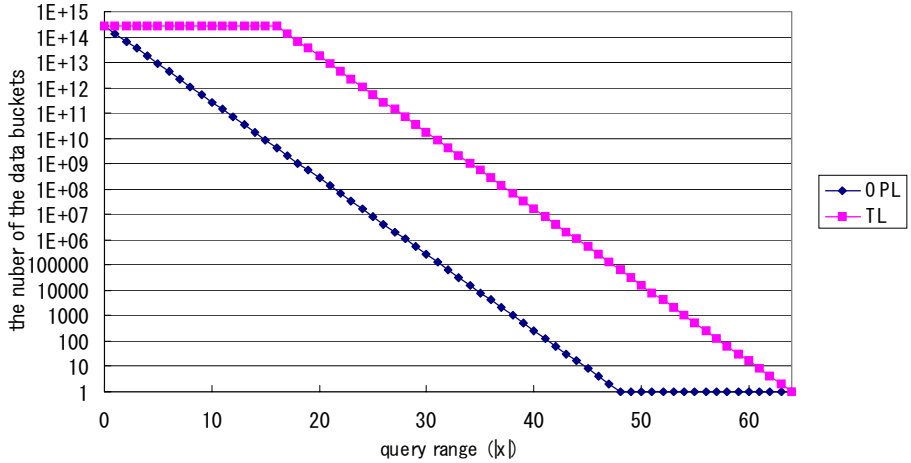

Figure 11. The Time Complexity (hash level = 64)

linear hashing[10]. Our scheme can be adapted to this distributed version of linear hashing, and can process a range query in parallel.

VII. CONCLUSIONS

We have proposed order-preserving linear hashing scheme enabling efficient retrieval for range queries. The hash function in the order-preserving linear hashing scheme is the combination of the division function and the bit reversal function. By using this function, the neighbor data are stored in the same data bucket in most cases. Therefore, the number of data buckets searched in the range query can be reduced. By comparing our hashing scheme with the traditional linear hashing scheme, the advantage of our hashing scheme was proved.

But the cost of the retrieval on our new linear hashing is higher than that on a B+tree in some cases. The future works include adapting our hashing scheme to the distributed version of linear hashing and the comparison of then.

REFERENCES

[1] Larson, P. A., "Dynamic Hashng", BIT ,vol. 18, 1978, pp. 184-201.

[2] R. Fagin, J.Njevergelt, N. Pippenger, And H. R. Strong, "Extendible hashing – a fast access method for dynamic files," ACM Trans. on Database Systems vol. 4, no. 3, 1979, pp. 315-344.

[3] W. Litwin, "Linear hashing: new tool for file and table addressing," Proc. Intle. Conf. on Very Large Databases, 1980, pp. 212-233.

[4] W. A. Burkhard, "Interpolation-based index maintenance," BIT, vol. 23, no. 3, 1983, pp.274-294

[5] J. T. Robinson, "Order preserving linear hashing using dynamic key statistics," Proc of the 5th ACM SIGACT-SIGMOD symposium on Principles of database systems, 1985, pp 91-99

[6] H. P. Kriegel, B. Seeger, "Multidimensional order preserving linear hashing with partial expansions," Proc. on International conference on database theory, 1986, pp.203-220.

[7] S. Lin, M. T. Ozsu, V. Oria, R. Ng "An extendible hash for multi-precision similarity querying of image databases," Proc. of 27th VLDB, 2001. pp 221-230.

[8] D. Tam, R. Azimi, H.A. Jacobsen "Building content-based publish/subscribe systems with distributed hash tables," Proc. of 1st DBISP2P2003, 2003, pp. 138-152.

[9] A. Andrzejak, Z. Xu "Scalable, efficient range queries for grid information services," Proc of 2nd P2P2002, 2002, pp. 33-40.

[10] W. Litwin, M. Neimat, and D.A.Schneider, "LH* - Linear Hashing for Distributed Files," ACM SIGMOD Conference, 1993, pp. 327-336.

.