Linear Hashing

Donghui Zhang
College of Computer & Information Science
Northeastern University
Boston, MA, USA
donghui@ccs.neu.edu

Yannis Manolopoulos
Department of Informatics
Aristotle University
Thessaloniki, 54124 Greece
manolopo@csd.auth.gr

Yannis Theodoridis
Department of Informatics
University of Piraeus
18534 Piraeus, Greece
ytheod@unipi.gr

Vassilis J. Tsotras
Department of Computer Science
and Engineering
University of California - Riverside
Riverside, CA 92521
tsotras@cs.ucr.edu

DEFINITION

Linear Hashing is a dynamically updateable disk-based index structure which implements a hashing scheme and which grows or shrinks one bucket at a time. The index is used to support exact match queries, i.e. find the record with a given key. Compared with the B+-tree index which also supports exact match queries (in logarithmic number of I/Os), Linear Hashing has better expected query cost O(1) I/O. Compared with Extendible Hashing, Linear Hashing does not use a bucket directory, and when an overflow occurs it is not always the overflown bucket that is split. The name Linear Hashing is used because the number of buckets grows or shrinks in a linear fashion. Overflows are handled by creating a chain of pages under the overflown bucket. The hashing function changes dynamically and at any given instant there can be at most two hashing functions used by the scheme.

HISTORICAL BACKGROUND

A hash table is an in-memory data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key, find the corresponding value. It works by transforming the key using a hash function into a hash, a number that is used as an index in an array to locate the desired location where the values should be. Multiple keys may be hashed to the same bucket, and all keys in a bucket should be searched upon a query. Hash tables are often used to implement associative arrays, sets and caches. Like arrays, hash tables have O(1) lookup cost on average.

SCIENTIFIC FUNDAMENTALS

The Linear Hashing scheme was introduced by [2].

Initial Layout The Linear Hashing scheme has m initial buckets labelled 0 through $m-1$, and an initial hashing function $h_0(k) = f(k) \% m$ that is used to map any key k into one of the m buckets (for simplicity assume $h_0(k) = k \% m$), and a pointer p which points to the bucket to be split next whenever an overflow page is generated (initially $p = 0$). An example is shown in Figure 1.

Bucket Split When the first overflow occurs (it can occur in any bucket), bucket 0, which is pointed by p, is split (rehashed) into two buckets: the original bucket 0 and a new bucket m. A new empty page is also added in the overflown bucket to accommodate the overflow. The search values originally mapped into bucket 0 (using

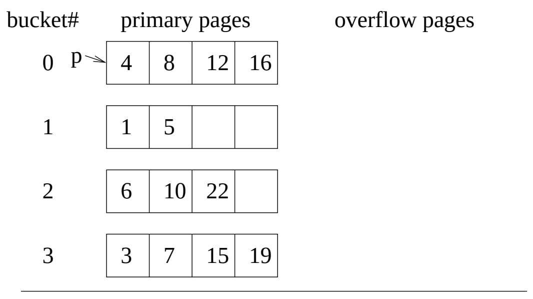

Figure 1: An initial Linear Hashing. Here $m = 4$, $p = 0$, $h_0(k) = k \% 4$.

function h_0) are now distributed between buckets 0 and m using a new hashing function h_1.

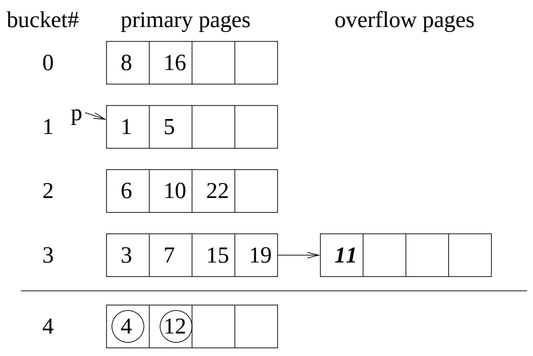

Figure 2: The Linear Hashing after inserting 11 into Figure 1. Here $p = 1$, $h_0(k) = k \% 4$, $h_1(k) = k \% 8$.

As an example, Figure 2 shows the layout of the Linear Hashing of Figure 1 after inserting a new record with key 11. The circled records are the existing records that are moved to the new bucket. In more detail, the record is inserted into bucket 11 % 4 = 3. The bucket overflows and an overflow page is introduced to accommodate the new record. Bucket 0 is split and the records originally in bucket 0 are distributed between bucket 0 and bucket 4, using a new hash function $h_1(k) = k \% 8$.
The next bucket overflow, such as triggered by inserting two records in bucket 2 or four records in bucket 3 in Figure 2, will cause a new split that will attach a new bucket $m+1$ and the contents of bucket 1 will be distributed using h_1 between buckets 1 and $m+1$. A crucial property of h_1 is that search values that were originally mapped by h_0 to some bucket j must be remapped either to bucket j or bucket $j + m$. This is a necessary property for Linear Hashing to work. An example of such hashing function is: $h_1(k) = k \% 2m$. Further bucket overflows will cause additional bucket splits in a linear bucket-number order (increasing p by one for every split).

Round and Hash Function Advancement After enough overflows, all original m buckets will be split. This marks the end of splitting-round 0. During round 0, p went subsequently from bucket 0 to bucket $m - 1$. At the end of round 0 the Linear Hashing scheme has a total of $2m$ buckets. Hashing function h_0 is no longer needed as all $2m$ buckets can be addressed by hashing function h_1. Variable p is reset to 0 and a new round, namely splitting-round 1, starts. A new hash function h_2 will start to be used.
In general, the Linear Hashing scheme involves a family of hash functions h_0, h_1, h_2, and so on. Let the initial function be $h_0(k) = f(k) \% m$, then any later hash function $h_i(k) = f(k) \% 2^i m$. This way, it is guaranteed that if h_i hashes a key to bucket $j \in [0..2^i m - 1]$, h_{i+1} will hash the same key to either bucket j or bucket $j + 2^i m$. At any time, two hash functions h_i and h_{i+1} are used.
Figure 3 and Figure 4 illustrates the cases at the end of splitting-round 0 and at the beginning of splitting-round 1. In general, in splitting round i, the hash functions h_i and h_{i+1} are used. At the beginning of round i, $p = 0$ and there are $2^i m$ buckets. When all of these buckets are split, splitting round $i + 1$ starts. p goes back to 0. the number of buckets becomes $2^{i+1} m$. And hash functions h_{i+1} and h_{i+2} will start to be used.

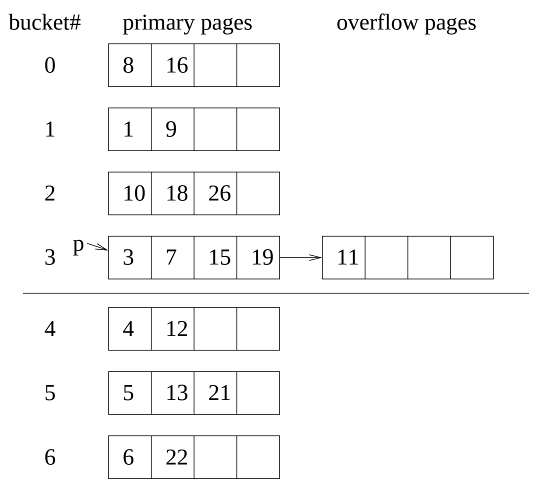

Figure 3: The Linear Hashing at the end of round 0. Here $p = 3$, $h_0(k) = k \ \% \ m$, $h_1(k) = k \ \% \ 2^1 m$.

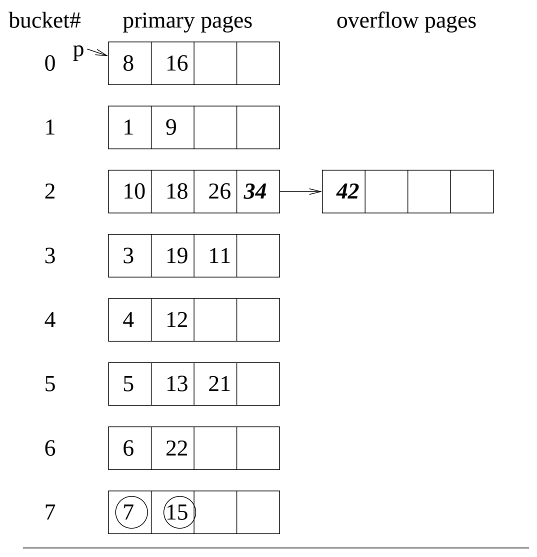

Figure 4: The Linear Hashing at the beginning of round 1. Here $p = 0$, $h_1(k) = k \ \% \ 2^1 m$, $h_2(k) = k \ \% \ 2^2 m$.

Component Summary and Search Scheme In summary, at any time a Linear Hashing scheme has the following components:

A value i which indicates the current splitting round.

- A variable $p \in [0..2^i m - 1]$ which indicates the bucket to be split next.
- A total of $2^i m + p$ buckets, each of which consists of a primary page and possibly some overflow pages.
- Two hash functions h_i and h_{i+1}.

A search scheme is needed to map a key k to a bucket, either when searching for an existing record or when inserting a new record. The search scheme works as follows:

If $h_i(k) \geq p$, choose bucket $h_i(k)$ since the bucket has not been split yet in the current round.

(b)If $h_i(k) < p$, choose bucket $h_{i+1}(k)$, which can be either $h_i(k)$ or its spit image $h_i(k) + 2^i m$.

For example, in Figure 2, $p = 1$. To search for record 5, since $h_0(5) = 1 \geq p$, one directly goes to bucket to find the record. But to search for record 4, since $h_0(4) = 0 < p$, one needs to use h_1 to decide the actual bucket. In this case, the record should be searched in bucket $h_1(4) = 4$.

Variations A split performed whenever a bucket overflow occurs is an uncontrolled split. Let l denote the Linear Hashing scheme's load factor, i.e., $l = S/b$ where S is the total number of records and b is the number of buckets used. The load factor achieved by uncontrolled splits is usually between 50-70%, depending on the page size and the search value distribution [2]. In practice, higher storage utilization is achieved if a split is triggered not by an overflow, but when the load factor l becomes greater than some upper threshold. This is called a controlled split and can typically achieve 95% utilization. Other controlled schemes exist where a split is delayed until both the threshold condition holds and an overflow occurs.
Deletions will cause the hashing scheme to shrink. Buckets that have been split can be recombined if the load factor falls below some lower threshold. Then two buckets are merged together; this operation is the reverse of splitting and occurs in reverse linear order. Practical values for the lower and upper thresholds are 0.7 and 0.9, respectively.
Linear Hashing has been further investigated in an effort to design more efficient variations. In [3] a performance comparison study of four Linear Hashing variations is reported.

KEY APPLICATIONS

Linear Hashing has been implemented into commercial database systems. It is used in applications where exact match query is the most important query such as hash join [4]. It has been adopted in the Icon language [1].

CROSS REFERENCE

Hashing, Extendible Hashing, Bloom Filter.

RECOMMENDED READING

[1] W. G. Griswold and G. M. Townsend. The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon. *Software: Practice and Experience*, 23(4):351–367, April 1993.

[2] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proc. 6th International Conference on Very Large Databases (VLDB)*, pages 212–223, 1980.

[3] Y. Manolopoulos and N. Lorentzos. Performance of Linear Hashing Schemes for Primary Key Retrieval. *Information Systems*, 19(5):433–446, 1994.

[4] D. A. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proc. 16th International Conference on Very Large Databases (VLDB)*, pages 469–480, 1990.