Performance of Linear Hashing and Spiral Hashing

Arockia David Roy Kulandai SJ
Dept. Computer Science
Marquette University
Milwaukee, WI, USA
david.roy@marquette.edu

Thomas Schwarz SJ
Dept. Computer Science
Marquette University
Milwaukee, WI, USA
thomas.schwarz@marquette.edu

***Abstract*—Linear Hashing is an important ingredient for many key-value stores. Spiral Storage was invented to overcome the poor fringe behavior of Linear Hashing, but after an influential study by Larson, seems to have been discarded. Since almost 50 years have passed, we repeat Larson's comparison with in-memory implementation of both to see whether his verdict still stands. Our study shows that Spiral Storage has slightly better look-up performance, but slightly poorer insert performance.**

I. Introduction

Key-value stores are a mainstay of data organization in Big-Data. Amazon DynamoDB is a pioneering NoSQL database built on this concept. A key-value store implements a *map* or *dictionary*. They can be implemented in different ways. B-tree like data structures allow for range queries, whereas dynamic hash tables have simpler architectures. For instance, Linear Hashing (LH) is used internally by both Amazon DynamoDB and BerkleyDB. These data structures served originally and continue to serve as indices for database tables. They were developed for a world of weak CPUs and smallish and slow storage systems, but have been successfully adapted to the world of Big-Data.

In this article, we compare LH with a competitor proposed at roughly the same time in the late eighties. LH stores key-value pairs in buckets. In its standard form, it uses a number of binary digits of the hash of the key to allocate the key-value pair to a bucket. At its inception, LH was implemented with buckets being stored in a single page or a block of pages in storage, then a magnetic hard drive. LH grows by splitting buckets in order, using more and more digits from the hash of the key to calculate the address. LH has always been criticized for cyclic worst case performance for both inserts and lookups. The ingenious addressing mechanism of *Spiral Storage* or *Spiral Hashing* (SH), as we prefer to call it, avoids this cyclic behavior. Like LH, SH stores key-value pairs in buckets. Its avoidance of cyclicity in its worst-case behavior comes at the price of a more complex address calculation. In a well-received study, [6], in the Communications of the ACM, Larson compared both LH and SH (as well as unbalanced binary tree and double hashing) and came to the conclusion that LH always outperforms SH. This experimental result probably reflects the greater complexity of SH addressing. Both data structures have changed by a considerable margin. More data is stored in memory or in the new NVRAM. Processor speeds and memory access times have changed. We decided to test whether Larson's verdict still stands. It does not!

In the following, we first review the standard versions of LH and SH. We then perform a theoretical Fringe analysis for bulk insertions into both data structures. Even if a large bulk of data is inserted, and thus a large number of splits occur, LH's performance is still cyclic. Then, we explain our implementations for LH and SH as main memory tables. The next section gives our experimental results and conclusions.

II. Background

Hashing schemes implement key-value stores and provide the key based operations of insert, update, delete, and read. A record consists of a key and a value. Hashing places the record in a location calculated from the key. Dynamic hashing schemes adjust to changes in the number of records so that the timing of the key based operations is almost or completely independent of the number of records. Most dynamic hashing schemes place records into buckets. They differ in the organization and behaviour of buckets. Originally, the classic dynamic hashing schemes stored records in disks. Nowadays, they are also used to distribute data over a distributed system or to store records in NVRAM, flash memory or in main memory.

A. Linear Hashing

Linear Hashing (LH) is a dynamic hashing scheme providing stable performance, good space utilization, and allows expansions and contractions of the LH file. It stores records in buckets which could be, but do not have to be, pages in a storage device. In the latter case, the bucket capacity is limited and overflow records would be stored in overflow buckets. LH has been widely used in disk oriented database systems such as, BerkleyDB and PostgreSQL [11].

The number of buckets in an LH file increases linearly with the number of elements inserted. A popular way maintains the load factor (i.e. ratio of number of elements over bucket) and sets the number of buckets to $\lfloor \alpha \cdot n \rfloor$, n being the number of records. Like Fagin's extendible hashing, the number of buckets increases through splits. Unlike Fagin's extendible hashing, the bucket to split is predetermined and independent of the size of buckets [4].

Internally, LH maintains a file state that in principle consists only of the current number b of buckets. Buckets are numbered starting with zero. From b we calculate the level l and the split pointer s as

$$b = 2^l + s,\ s < 2^l.$$

If the number of buckets is incremented, Bucket s is split into Bucket s and Bucket $s + 2^l$. In this case, the split pointer s is incremented. If s is equal to 2^l, then the level l is incremented and s reset to zero. To calculate the bucket in which a record with key c resides, LH uses a family of "consistent hash functions". One way to implement such as family is to use a single hash function h yielding many bits and define partial hash functions h_i by $h_i(c) = h(c) \pmod{2^i}$, i.e., the last i bits of $h(c)$. If the LH file has level l and split pointer s, the partial hash functions h_l and h_{l+1} are used. To be more precise, the address of a record with key c, i.e., the bucket in which the record should reside is first calculated as $a = h_l(c)$, but if $a < s$, recalculated as $a = h_{l+1}(c)$.

Interestingly, the bucket that splits is not necessarily the bucket that overflows. This makes the algorithm very simple. An alternative is Fagin's extendible hashing that always splits an overflowing bucket. The price to pay is a special directory structure. In LH [7], buckets are split according to the scheme

$$0; 0, 1; 0, 1, 2, 3; 0, 1, 2, 3, 4, 5, 6, 7; \cdots .$$

During a split all records in the bucket to split are rehashed and either remain in the same bucket or moved to the new bucket.

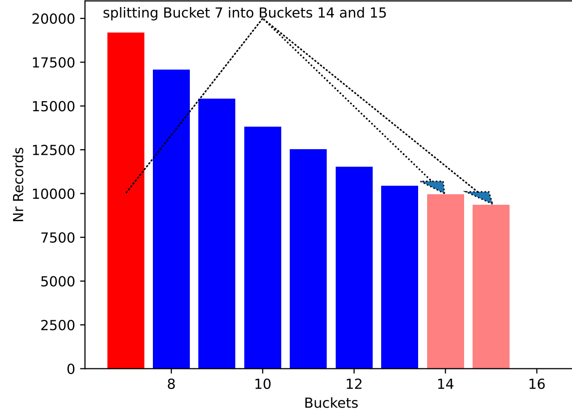


Fig. 1. Spiral Hashing with S=7, before and after splitting.

```
def address(key, S):
    s = math.log(S,2)
    x = math.ceil(s-key)+key
    y = math.floor(2**x)
    return y
```

Fig. 2. Address calculation in Spiral Hashing for key `key` and file state `S`.

B. Spiral Hashing

A drawback to LH is the existence of non-split and already split buckets. The former tend to have twice as many records than the latter. As we will see, Section III, the number of records rehashed when a file is split depends on the size of the split pointer and varies between $1/\alpha$ and $2/\alpha$. To avoid this behaviour, Spiral Hashing (SH) was invented by Martin [2, 8, 9]. Spiral Hashing intentionally distributes the records unevenly into Buckets S, $S + 1$, ..., $2S - 1$ where S is the file state, Figure 1. When the file grows, the contents of Bucket S are distributed into two new Buckets $2S$ and $2S + 1$. Afterwards, Bucket S is deleted. The probability of a record ever been allocated to Bucket i is $p_i = \log_2(1+1/i)$. This remarkable feat is achieved with a logarithmic address calculation given in Figure 2. SH can easily be modified to generate d new buckets for each freed one. Larson [6] in 1988 came to the conclusion that LH always outperforms SH. Since externalities such as typical uses, typical sizes, and typical platforms have changed since, we repeat the experimental evaluation and comparison of both.

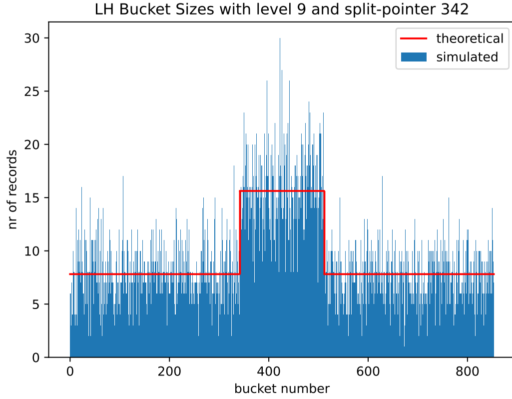


Fig. 3. Number of records per bucket in an LH file with level 9 and split pointer 342.

III. Fringe Analysis

Fringe analysis [1] analyzes the behavior of a data structure under mass insertion. For instance, Glombiewski, Seeger, and Graefe showed that a steady stream of inserts into a B-tree can create "waves of misery", where rebuilding leads to waves of data movements [5]. While Linear Hashing is alluring because of its conceptual and architectonic simplicity, its buckets during an epoch without change of level, i.e. with buckets in the half-open interval $[2^l, 2^{l+1})$, l being the level, fall into two different categories. A bucket in an LH-file with file state level l and split pointer s before being split, i.e. with a bucket number between $2^l + s$ and 2^{l+1} is twice the size of a bucket after a split, i.e. with a bucket number between 2^l and 2^l+s or between 2^{l+1} and $2^{l+1}+s$, Fig. 3. There we show the actual and theoretically expected number of records in one of the 854 buckets of an LH file with $8,000$ records.

If the LH-file has a load factor of α and n records, then there will be $\alpha \cdot n$ buckets. The level is therefore $l = \lfloor \log_2(\alpha \cdot n) \rfloor$. An unsplit bucket absorbs on average 2^{-l} of all records and has therefore $2^{-l}\alpha \cdot n$ records on average. The average size of a bucket is $1/\alpha$ as there are n records and $\alpha \cdot n$ buckets. If there are $(2^x + 1)/\alpha$ records, LH will have just moved to 2^x buckets and incremented the level to x. The first split (after growing to $\frac{2^x}{\alpha} + \lceil \frac{1}{\alpha} \rceil$) will move on average $\frac{1}{\alpha} + \frac{2^{-l}}{\alpha}$. On the other hand, the last bucket split before the level is incremented is with $2^{x+1}\alpha^{-1} - \alpha^{-1}$ records, and now the bucket to be split has $2^{-l}(\frac{2^{l+1}}{\alpha} - \frac{1}{\alpha})$ records in it. The ratio between these two numbers is $2 - \frac{3}{1+2^l}$ which converges quickly to 2. With growing i data movemement and the number of additional keys hashed with a single insert triggereing a split follows a see-saw curve with the lower value being half the higher value. The same happens when we have a batch insert, though the differences are slightly less. In Figure 4, we show the number of records in an LH file starting at n records, $n \in [10000, 20000]$, that are evaluated when 1000 records (a growth between 5% and 10%) are added.

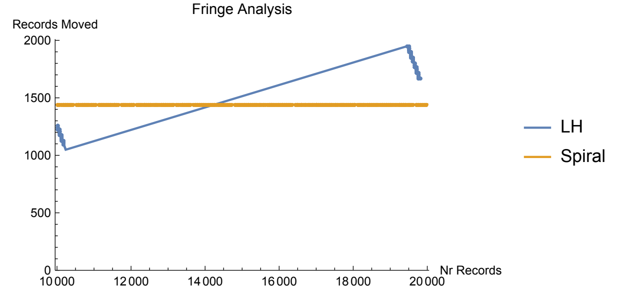


Fig. 4. Number of records rehashed when adding 1000 records to a Linear Hashing and Spiral Hashing file with n records, $n \in [10000, 20000]$.

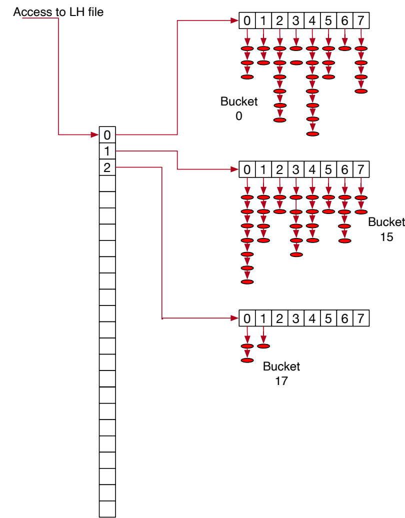


Fig. 5. Bucket organization in Linear Hashing with a records arranged as singly linked-lists.

Spiral Hashing's selling point is the avoidance of this fringe behavior. The probability of a record belonging to Bucket i is $\log_2(1 + \frac{1}{i})$. The size of the bucket that is split is always rather close to $\frac{1.44}{\alpha}$. Thus, instead of a cyclic fringe behavior, Spiral Hashing shows close to constant movement, independent of the number of records, Figure 4.

IV. Implementations

We implement both LH and SH in line with Larson and Ellis Schlatter's recommendations as an in-memory

structure [3, 6]. The records are arranged in buckets in two-stage dynamic arrays of linked-lists, Figure 5. Pointers to contiguous sets of buckets are collected in a segment array. Each segment array has a fixed number of pointers. A master array then collects pointers to segments. This implements a dynamic array with little memory overhead. Segments are created when needed and, in case of SH, deleted when unused. The range of buckets in an LH or SH file is always contiguous. Only the last segment and the first segment in case of an SH file may be partially used. The unused area of the directory array is relatively small. If the directory array is full, it is replaced by one twice its size with the same non-null entries. In our implementation, we used the vector data structure from C++17 standard library. Buckets are singly linked-lists in which we insert at the head.

Each bucket has a lock and so does each bucket segment. There is also a global lock for the file state. Deadlock is avoided because locks are only acquired in order from file state lock to a bucket lock. Locks can be exclusive and shared. We use exclusive locks for inserts and inclusive locks for lookup and update operations.

For our experiments, we created an LH and SH data structures with bucket target capacities of three, five and ten records, corresponding to load factors $1/3$, $1/5$, and $1/10$. We tested behaviour of insertions and lookups independently. For insertions, we generated one million records and inserted them dividing the work among a varying number of threads. To avoid the contention for buckets in a small file, we also measured the runtime after insertion of a different set of one million records. For the lookup, we shuffled the array of records inserted and then divided the lookups among various numbers of threads.

V. Measurements

Figures 6, 7, and 8 give our results. They were obtained from MacBook Pro with 2.4 GHz 8-Core Intel i9 processor. The y-axis gives the total execution time. We give both the minimum and the average time of twenty independent runs. Not surprisingly, the larger the load factor the slower the inserts and the faster are the lookups. For inserts, spreading the work among more threads lowers the performance. We can attribute this to the contention for file state and bucket segments as frequent splits occur. Preloading has almost no effect. Presumably, the file stays small for a comparatively short time so that concurrent inserts are being made to different buckets almost always. Lookups benefit from

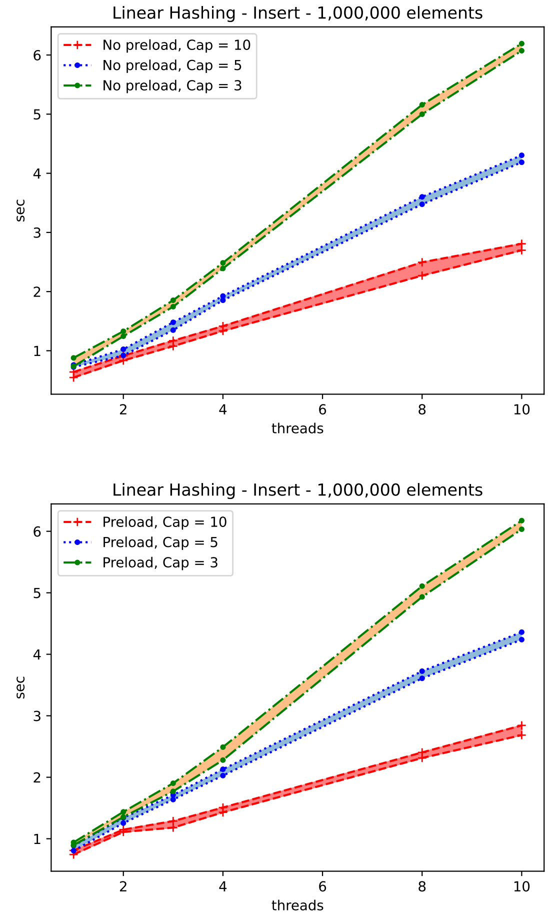


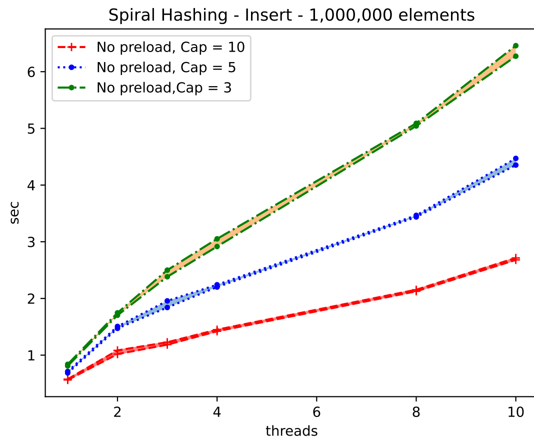
Fig. 6. Linear Hashing insert with and without preload.

distributing the work among different threads but only moderately so.

Spiral Hashing slightly outperforms Linear Hashing for lookups but looses slightly for inserts. We can attribute this to the architecture of SH which needs more rebuild for splits, namely, the creation of two new buckets and the deletion of one old bucket per split. Internally, the bucket splits are implemented in the same manner. Even for LH, a split creates two new buckets and deletes the old one, but bucket segments are more likely to be added in SH and only SH deletes bucket segments. For lookups, the existence of large buckets in LH seems to explain the slightly longer timing. Even with a nominal capacity of ten, an un-split LH bucket can have on average almost twenty records.

VI. Conclusions and Future Work

Key-value stores are a mainstay technology of Big-Data. The underlying data structures were invented some 40 years ago in a different world. Linear Hashing is used



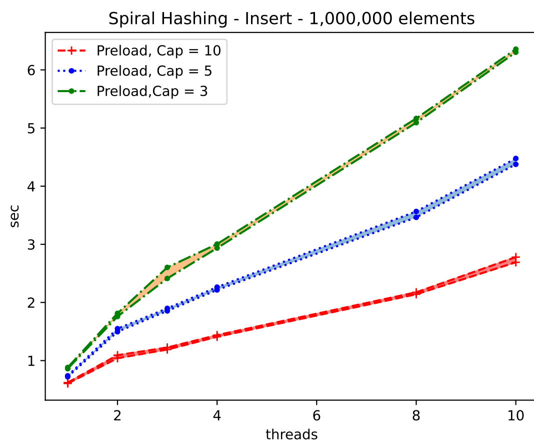
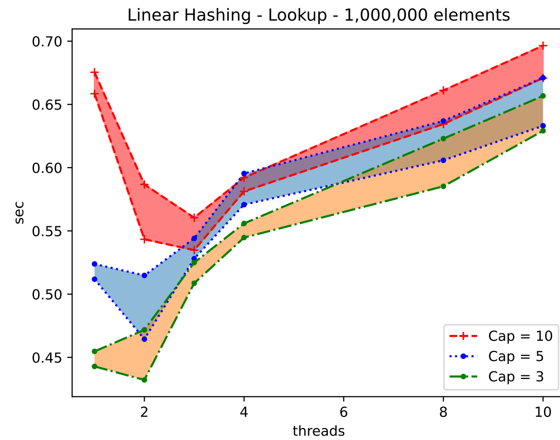


Fig. 7. Spiral Hashing with and without preload.



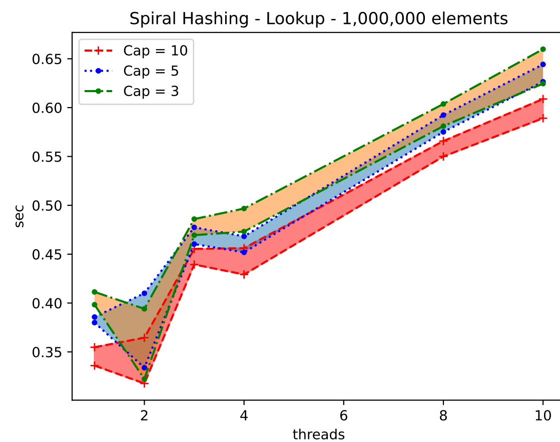


Fig. 8. Linear Hashing and Spiral Hashing lookups.

extensively. To address a perceived downside of Linear Hashing, Spiral Storage was proposed. A seminal article by Larson in the Communications of the ACM showed the superiority of Linear Hashing and work on Spiral Hashing dried up. Presumably, the difference observed by Larson is due to the more involved address calculation in Spiral Hashing. In this paper, we repeated his comparison and found that the performance differences are small and that Spiral Hashing might actually be sightly advantageous.

Our implementation used locks. Lockfree implementations of Linear Hashing [10, 12] exist and one for Spiral Hashing might be simpler because the old bucket after a split is no longer used, but can remain accessible for a time. This is a promising research direction. Second, a distributed version of Spiral Hashing can select the two best servers for the new buckets instead of being forced to live with the placement of the old bucket as happens in Linear Hashing. This constitutes another promising direction for future work.

References

[1] R. A. Baeza-Yates, "Fringe analysis revisited," *ACM Computing Surveys (CSUR)*, vol. 27, no. 1, pp. 109–119, 1995.

[2] J.-H. Chu and G. D. Knott, "An analysis of spiral hashing," *The Computer Journal*, vol. 37, no. 8, pp. 715–719, 1994.

[3] C. S. Ellis, "Concurrency in linear hashing," *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 2, pp. 195–217, 1987.

[4] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing—a fast access method for dynamic files," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.

[5] N. Glombiewski, B. Seeger, and G. Graefe, "Waves of misery after index creation," *Datenbanksysteme für Business, Technologie und Web, Lecture Notes*

in Informatics, Gesellschaft für Informatik, Bonn, pp. 77–96, 2019.
[6] P.-A. Larson, "Dynamic hash tables," *Communications of the ACM*, vol. 31, no. 4, pp. 446–457, 1988.
[7] W. Litwin, "Linear hashing: a new tool for file and table addressing." in *Proceedings, Conference on Very Large Database Systems*, 1980.
[8] G. N. N. Martin, "Spiral storage: Incrementally augmentable hash addressed storage," *Theory of Computation Report - CS-RR-027*, 1979.
[9] J. K. Mullin, "Spiral storage: Efficient dynamic hashing with constant performance," *The Computer Journal*, vol. 28, no. 3, pp. 330–334, 1985.
[10] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM (JACM)*, vol. 53, no. 3, pp. 379–405, 2006.
[11] H. Wan, F. Li, Z. Zhou, K. Zeng, J. Li, and C. J. Xue, "NVLH: crash-consistent linear hashing for non-volatile memory," in *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2018, pp. 117–118.
[12] D. Zhang and P.-Å. Larson, "Lhlf: lock-free linear hashing (poster paper)," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 307–308.