

Optimality in External Memory Hashing

Morten Skaarup Jensen* and Rasmus Pagh*

April 2, 2007

Abstract

Hash tables on external memory are commonly used for indexing in database management systems. In this paper we present an algorithm that, in an asymptotic sense, achieves the best possible I/O and space complexities. Let B denote the number of records that fit in a block, and let N denote the total number of records. Our hash table uses $1 + O(1/\sqrt{B})$ I/Os, expected, for looking up a record (no matter if it is present or not). To insert, delete or change a record that has just been looked up requires $1 + O(1/\sqrt{B})$ I/Os, amortized expected, including I/Os for reorganizing the hash table when the size of the database changes. The expected external space usage is $1 + O(1/\sqrt{B})$ times the optimum of N/B blocks, and just $O(1)$ blocks of internal memory are needed.

Key words: External memory, dictionary, hashing, index

1 Introduction

External memory hashing is a well known, efficient solution to the problem of storing a set of records, each containing a unique key, such that the record with a certain key can be found quickly (this is also known as the *dictionary* problem). We also consider *update* operations that change the set of records: Insertion of a new record (whose key did not exist in any record before), changing an existing record, and deleting an existing record.

We will assume that a record has fixed size, which is not a serious restriction since records could contain pointers to associated information of variable size. The question addressed in this paper is how efficiently an external memory hash table can be implemented. The model of computation considered is the classical I/O model of Aggarwal and Vitter [1]. Let B denote the number of records that fit in a block of external memory. We show that as B grows it is possible to approach, simultaneously, optimal space, update, and query complexity. More precisely, if N is the total number of records, the expected space usage is $(1 + O(1/\sqrt{B})) N/B$ blocks of external memory, and $O(1)$ blocks of internal

*IT University of Copenhagen, Rued Langgaards Vej 7, 2300 København S, Denmark. {mskaarupj, pagh}@itu.dk

memory. Note that a space usage of N/B blocks would be optimal. The expected number of I/Os to look up a record with a given key is $1 + O(1/\sqrt{B})$, even in the case where there exists no record with that key. Update operations start with a lookup of the relevant key, leaving the block to which the key is mapped in internal memory. After this, the cost of performing the update is $1 + O(1/\sqrt{B})$ I/Os, amortized expected. This bound includes I/Os for reorganizing the hash table when the size of the set changes. The I/O bound for insertions and record changes is optimal in the sense that 1 write I/O is needed to commit any change to secondary memory, as is often desired in applications of external memory hash tables.

One interpretation of our result is that as technology develops and external memory blocks get bigger and bigger, our data structure approaches the optimum of N/B external memory blocks, 1 I/O for updates, and 1 I/O for lookups. In this context it should be mentioned that in recent years the logical block size of disks has not grown proportionally with the increase in data density, probably due to the fact that many disk blocks are used for small amounts of data (a small file, say), which means that larger blocks lead to poorer space utilization. However, in our data structure the space utilization improves as the block size grows, so it makes sense to use large logical blocks, as long as the time to make an I/O is dominated by the seek time.

As presented in this paper, our result is a theoretical one, establishing bounds of the form $1+o(1)$ for all main performance parameters. Practical use would require a closer look at the constants in the lower order terms. This appears to be a somewhat complicated task, since there are (perhaps unavoidable) trade-offs between space overhead, lookup operations, and update operations.

1.1 Assumptions and terminology

As in other works on external memory hashing we will work with the model that we have access to a hash function h that maps the set of keys to random and independent values. We use S to denote the set of all keys of records in the hash table. For convenience, we will interchangeably refer to records and their keys (e.g., “inserting the key x ” implies that we insert the corresponding record). A *bucket* is a subset of S that has a given value of h . We will refer to the number of keys in a bucket as its *load*.

External memory hash tables typically have the following basic structure (also used in this paper): There is a hash table, containing most records. If possible, the bucket with hash value i is stored in block number i of the hash table (referred to as the *primary block* of the bucket). Otherwise, if the size of the bucket exceeds B , we say that the bucket is *overflowing*. Overflowing buckets must have some records stored in a secondary structure, typically a linked list. These records are referred to as *overflow records*.

1.2 Related work

If sufficient internal storage is available, it is possible to do hash table lookups in 1 I/O in the worst case [5]. However, this requires internal storage that grows linearly with N , so

it is not feasible for large data sets. From now on we will consider only hashing schemes whose internal space usage is $O(1)$ blocks. We will see that this very limited amount of internal storage suffices to get arbitrarily close to optimal bounds as B grows.

It is known that it is possible to get arbitrarily close to 1 I/O for lookups and updates, for large B , if the space utilization is fixed to some constant smaller than 1, see e.g. [7]. If an upper bound on the size of the set is fixed, the variant of *linear probing* in which probe sequences always start at the beginning of a block can be shown to achieve $1 + o(1)$ I/Os, expected, for all operations, and space within a factor $1 + o(1)$ of optimal. The lower order terms, which are functions of B , decrease asymptotically more slowly than the term of $O(1/\sqrt{B})$ achieved in this paper. Also, it is unclear whether this would generalize to the case in which the hash table needs to expand in response to an increase in the number of keys.

The space complexity is important also in terms of time, because it is reflected in the time it takes to scan the whole hash table (a common operation in database systems). It is conceivable that the amount of space used also affects the time it takes to make an I/O.

There have been many proposals for handling overflows in hash tables, the goal being to minimize the probability of a bucket overflowing, and handling the overflow records in an efficient way (e.g. [4, 6, 8, 10, 11, 14, 15]). When a new record is inserted in an overflowing bucket, the common solution is to place it in an overflow area. This means that all subsequent lookups of that record require at least 2 I/Os. The same is true in this case when unsuccessfully searching for a record with a key that does not exist in the set. If the space utilization is a factor $1 + O(1/\sqrt{B})$ from optimal this is quite serious, because we expect a constant fraction of the buckets to overflow, see [7, p. 541–545]. This means that unsuccessful lookups will use $1 + \Omega(1)$ I/Os, expected. One approach to minimizing search time that has been proposed is to store the records sorted by key. While this brings down the average unsuccessful search time, it has the same weaknesses as before when searching for keys that come late in the ordering.

Resizing the hash table. Another line of work has dealt with how to efficiently grow and shrink a hash table as the number of records increases or decreases, see [12]. To provide the necessary knowledge to understand our construction, we sketch a particular scheme that will suffice for our purposes, namely linear hashing with partial expansions (due to Larson [9], based on the seminal work of Litwin [13]). The blocks of the hash table are partitioned into groups, such that the size of any two groups differs by at most 1. The hash function values are distributed uniformly on the groups (i.e., the probability to have a hash function value in a particular group is the same for all groups), and are also distributed uniformly on the blocks within a group. To increase the number of blocks in the hash table by 1, the number of blocks in one of the groups is increased by 1, and the corresponding buckets are reorganized by hashing to the larger range. Decreasing the number of blocks in the hash table by 1 is done analogously. We will refer to the period between two points of time where all groups have the same size as a *phase*. At the beginning of each phase, we may split all groups into two groups, or merge pairs of groups in order to maintain a

group size of $\Theta(n_0)$, for some parameter n_0 . (In fact, this can be done without rearranging the keys, but this optimization is not essential for the present paper.) Linear hashing with partial expansions keeps the expected load on all buckets within a factor $1 + O(1/n_0)$ of each other at the amortized expected cost of $O(n_0/B)$ I/Os per update operation. It seems that in the literature n_0 is thought of as a constant; however, in this paper we will choose $n_0 = \Theta(\sqrt{B})$.

1.3 Technical contributions

To obtain our results, we introduce several new techniques. Though we only demonstrate their merits asymptotically for large B , they also seem to be practically useful for small B .

Diligent load balancing. Compared to previous algorithms, we use a factor $O(\sqrt{B})$ more time on expanding/contracting the hash table, to keep the load of buckets extremely stable.

Buffering to speed up insertions. Most insertions are done directly in the primary block, which has a “buffer” – a special part of the block with space for $O(\sqrt{B})$ records. Only when the buffer is full, we propagate records on to the overflow area.

Hash-based stop criterion. We use a hash function σ to determine which keys to store in the overflow area. A search for x does not go to the overflow area if $\sigma(x)$ is smaller than the σ value of some key in the primary block (except the buffer). This speeds up unsuccessful searches.

Duplication to utilize buffer space. Unused buffer space is utilized by duplicating some records in the overflow area. Because of the duplication, such records can be overwritten without any problem.

The last technique is merely an optimization, but the first three all seem essential to obtaining our result. The idea of using significantly more I/Os on keeping the load very stable, balancing the work spent on resizing the hash table with the overhead stemming from overflowing buckets, does not seem to have appeared in the literature. Also the hash-based stop criterion seems novel, though related ideas have been described [3, 4, 5]. Buffering is a well-known technique, previously used, e.g., in the context of B-trees [2], but to the best of our knowledge not in the context of hash tables.

Our analysis sheds light on a fundamental quantity, namely the expected number of overflow records in a bucket. Surprisingly, in the literature on hashing we have not been able to find any formal analysis of this quantity, which is related to the concept of mean deviation in descriptive statistics.

2 Our data structure

Our main idea is to reserve a special part of the primary block, with space for $\Theta(\sqrt{B})$ records, to use as a *buffer*. Most insertions are simply written to the buffer — only when

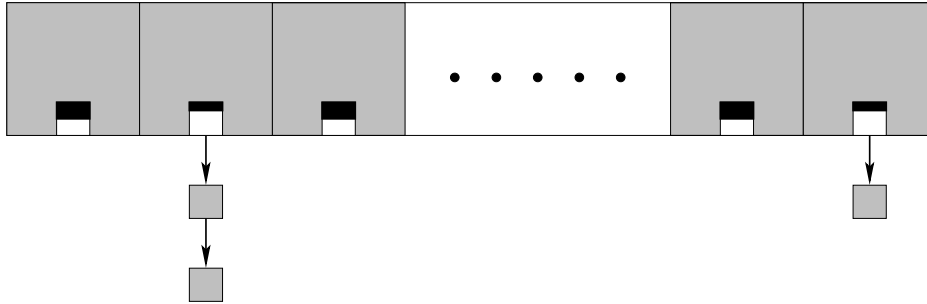


Figure 1: Data structure at a glance. Large boxes are blocks with capacity for B records, small boxes (buffers in primary blocks and linked lists of “blocklets”) have capacity for \sqrt{B} records. New records are first inserted in a primary block buffer, which is “flushed” when it becomes full.

the buffer becomes full we use additional I/Os to perform the updates by distributing the keys in the bucket in the non-buffer part of the primary block and a linked list of overflow records. The non-buffer part of the primary blocks has capacity for $c = B - O(\sqrt{B})$ records.

If there are more than c keys in a bucket, not counting keys stored in the corresponding buffer, there is a choice of which keys to store in the primary block, and which keys to store in the overflow chain. To address this, we use a hash function σ with range of size $\Omega(B^2)$ in addition to the primary hash function. (Note that both h and σ can be derived from a single hash function with range of size $\Omega(NB)$, so having two hash functions is not a stronger requirement than usual in this context.)

Our data structure maintains the following invariants:

1. The expected load of every bucket is at least $B - O(\sqrt{B})$ and at most B .
2. The non-buffer part of a primary block contains the records of keys that are smallest according to the order induced by σ (breaking ties arbitrarily). In particular, for any key x in the non-buffer part of a primary block and any key y in the corresponding overflow chain, we have $\sigma(x) \leq \sigma(y)$.

The choice of buffer size is subject to a trade-off between space usage (the buffer usually has a large unused part) and how often the buffer needs to be flushed. Choosing buffer size \sqrt{B} gives the coefficient $1 + O(1/\sqrt{B})$ on the update time as well as the space usage, but this is just one point on the trade-off curve.

We first describe the data structure under the additional invariant that a record appears exactly once in the data structure. In Section 2.3 we describe an optimization that makes use of duplication of records.

To store the overflow records of a primary block, we use a linked list of “blocklets”, i.e., parts of a disk block with a capacity of $\Theta(\sqrt{B})$ records. A blocklet can be read or written in 1 I/O. The details of memory management for the blocklets use standard techniques,

and allow the whole data structure to reside in a single contiguous segment of external memory.

The first invariant is maintained by linear hashing with partial expansions [9] with parameter $n_0 = C\sqrt{B}$, for some constant $C = O(1)$. To simplify our analysis, we require that during a phase we perform only expansions or only contractions of the table. Thus, a phase has the effect of increasing or decreasing the size of the hash table by a factor $1 + \Theta(1/\sqrt{B})$. This requirement only affects the constant C that must be used to maintain the first invariant. The cost of linear hashing with partial expansions is $O(n_0)$ I/Os for every $O(B)$ updates, so the amortized cost per update is $O(1/\sqrt{B})$ I/Os.

2.1 Implementation of operations

Lookup.

A lookup operation for a key x is implemented as follows: We first get the primary block $h(x)$, using 1 I/O. If the record we are looking for is there, it is returned. Otherwise we compare $\sigma(x)$ with the largest value of σ for a key in the primary block (ignoring keys in the buffer). If $\sigma(x)$ not smallest, we continue the search in the overflow chain. By the second invariant, we will always find the record containing x if it exists.

Insertion.

An insertion of a record with key x proceeds by writing the record to the buffer in block $h(x)$, if there is space. The assumption that there has just been a lookup of the key means that the relevant block resides in internal memory, so this requires 1 write I/O. If the buffer is full we rebuild the primary block and overflow area, such that the primary block is filled with the records of keys with the smallest σ -values, and the remaining records are stored in the overflow chain. The buffer is left empty (however, see Section 2.3). This requires $O(1)$ I/Os for each blocklet in the overflow chain.

Deletion.

Deletion of a record is simply done by rewriting the block in which it resides, using a single write I/O. If the number of records deleted from a bucket since the last rebuild exceeds $O(\sqrt{B})$, we rebuild the primary block and overflow area, as described above.

Change.

A change to a record is done by rewriting the block in which it resides, using 1 write I/O. Note that we are assuming that changes happen only to non-key attributes.

2.2 Analysis

Lookup.

We first consider the expected number of I/Os for a lookup operation. If the record of the key, x , has been inserted since the last rebuilding of bucket $h(x)$ it is found in 1 I/O. Otherwise consider the current set S of keys. We consider the random variable X denoting the number of keys that were present in bucket $h(x)$ when it was last rebuilt.

If $X \leq c$ there are no overflow records, so all keys are in the primary block (some may be in the buffer), and only 1 I/O will be used. If $X > c$ then the probability that the lookup of x continues in the overflow area is at most $\frac{X+1-c}{X+1} = O(\frac{X-c}{c})$, given that x does not collide with any other key in the bucket under σ . (We could eliminate this possibility by choosing σ as a random permutation, but having such a function would be a stronger assumption than just having an independent hash function.) We can make the collision probability arbitrarily low, and for the particular choice of range stated above, the probability is $O(1/B)$. The cost of searching the overflow area is $O(1 + \frac{X-c}{\sqrt{B}})$ I/Os. Hence, the expected cost of searching the overflow area in a lookup operation is

$$\begin{aligned} & O\left(\sum_{i>c} \Pr[X = i] \left(\frac{i-c}{c} + \frac{1}{B}\right) \left(1 + \frac{i-c}{\sqrt{B}}\right)\right) \\ &= O\left(\mathbf{E}[|X - c|]/c + \mathbf{E}[(X - c)^2]/(c\sqrt{B})\right) . \end{aligned}$$

The random variable X does not seem to have a “nice” distribution. However, because of the way insertions and deletions are implemented, it closely follows the random variable X' denoting the number of keys of S hashing to $h(x)$: $|X' - X| = O(\sqrt{B})$. This means that replacing X by X' in the above expression decreases its value by at most $O(1/\sqrt{B})$. Similarly, since $|\mathbf{E}[X'] - c| = O(\sqrt{B})$ (by invariant 1) we can replace c by $\mathbf{E}[X']$, decreasing the value of the expression by at most $O(1/\sqrt{B})$. Hence, it suffices to bound

$$\mathbf{E}[|X' - \mathbf{E}[X']|]/c + \mathbf{E}[(X' - \mathbf{E}[X'])^2]/(c\sqrt{B}) .$$

By the assumption that hash function values are independent, X' has a binomial distribution. The expression $\mathbf{E}[|X' - \mathbf{E}[X']|]$ is known as the *mean deviation* in descriptive statistics. We have been unable to find a bound on the mean deviation of a binomially distributed random variable in the literature, but show the following lemma.

Lemma 1 *For a binomially distributed random variable X , $\mathbf{E}[|X - \mathbf{E}[X]|] < \sqrt{\mathbf{E}[X]}$.*

Proof. Let $Z = |X - \mathbf{E}[X]|$. We have $0 \leq \mathbf{Var}(Z) = \mathbf{E}[Z^2] - \mathbf{E}[Z]^2 = \mathbf{Var}(X) - \mathbf{E}[Z]^2$. Hence $\mathbf{E}[Z] \leq \sqrt{\mathbf{Var}(X)} < \sqrt{\mathbf{E}[X]}$. The last inequality uses that X is binomially distributed. \square

By the lemma, the mean deviation of X' is $O(\sqrt{\mathbf{E}[X']})$, which implies that $\mathbf{E}[|X' - \mathbf{E}[X']|]/c = O(1/\sqrt{B})$. The expression $\mathbf{E}[(X' - \mathbf{E}[X'])^2]$ is simply the variance of X' ,

which is bounded by $\mathbf{E}[X'] \leq B$. Hence, we have $\mathbf{E}[(X' - \mathbf{E}[X'])^2]/(c\sqrt{B}) = O(1/\sqrt{B})$. In conclusion, the expected cost of searching the overflow area during a lookup is $O(1/\sqrt{B})$ I/Os.

Updates and table resizing.

Change operations always require 1 I/O. Insertions and deletions require 1 I/O unless there have been $\Theta(\sqrt{B})$ insertions or deletions in bucket $h(x)$ since it was last rebuilt. In this case the insertion or deletion procedure rebuilds the bucket. Let X be the number of keys hashing to $h(x)$. Then the number of blocklets in the overflow chain is $O(1 + |X - c|/\sqrt{B})$, so the cost of rebuilding is $O(1 + |X - c|/\sqrt{B})$ I/Os. By Lemma 1 the expected number of I/Os is thus $O(1)$ in this case. Amortized over the $\Theta(\sqrt{B})$ insertions or deletions the cost is thus $O(1/\sqrt{B})$ I/Os.

We will now account for the work that goes into resizing the hash table. Consider a phase of linear hashing with partial expansion, starting with N keys. During the $O(N/\sqrt{B})$ update operations of the phase, all records are read and written exactly once. To bound the cost of a phase we consider the set of at most $N + O(N/\sqrt{B})$ keys that exist at some point during the phase. The expected number of records in the overflow area is $O(N/\sqrt{B})$, and hence the cost of reading and writing all records is $O(N/B)$ I/Os. This is $O(1/\sqrt{B})$ I/Os, amortized, per update.

Space usage.

To account for the space usage, we sum up the total space not used for records. Our description of the data structure has made use of small amounts of non-record information, e.g. pointers. It is not hard to see that the space for this can be ignored. Alternatively, we sketch in Section 2.3 how to avoid this extra information.

As previously noted, the expected number of records in the overflow area is $O(N/\sqrt{B})$. Furthermore, the unused space in each of the $O(N/B)$ overflow chains and buffers corresponds to the space of $O(\sqrt{B})$ records, so this is $O(N/B^{3/2})$ blocks in total. Finally, the expected amount of unused space in each primary block is $O(\sqrt{B})$ records by invariant 1 and Lemma 1. In conclusion, the unused space is $O(N/B^{3/2})$ blocks, expected.

2.3 Optimizations

We have described our data structure in the simplest possible way that allowed us to prove our main result. There are several optimizations that can be made to improve the constant factor in the big-O.

Improved buffers.

To utilize unused buffer space one could make duplicates of the records with the smallest σ -values in the overflow chain. In this way, there will always be B records in the primary block (if there are at least B records in the bucket), and thus the space for the buffer is

not wasted. Secondly, deletions in a primary block can be exploited to increase the size of the buffer in the block.

Avoiding non-record information.

Instead of using linked lists to implement the overflow chains, we can use a standard hash table. To get the first blocklet of the chained list of bucket i one would look up the key $(i, 1)$, etc.

3 Conclusion and open problems

We have studied the fundamental question of the complexity of external memory hash tables in an asymptotic sense, showing that for large block size B one can get arbitrarily close to 1 I/O per operation, and optimal space. This is best possible if all changes must be committed to disk before an operation is considered to be complete.

It remains open how fast the convergence is, i.e., whether the factor $1 + O(1/\sqrt{B})$ is the best possible overhead. Our techniques can be used to obtain a trade-off with worse space usage but faster operations, but it seems reasonable to conjecture that the space overhead must be a factor $1 + \Omega(1/\sqrt{B})$ to achieve $1 + O(1/\sqrt{B})$ I/Os per update operation. An interesting open problem is to give an explicit way of choosing h such that the analysis goes through, still using only $O(1)$ blocks of internal memory. For example, is an $O(B)$ -wise independent family of hash functions sufficient, for all values of B ? Another open problem is to investigate whether the bound of 1 I/O for insertions and updates is optimal even if one allows operations to be cached in internal memory (i.e., not “committed” to disk straight away). We conjecture that this is indeed optimal if lookups are to be possible in $O(1)$ I/Os.

Acknowledgement. We thank the reviewers for their help in improving the exposition, and one reviewer for suggesting an improved proof of Lemma 1.

References

- [1] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.
- [2] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.
- [3] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. System Sci.*, 60(3):630–659, 2000.
- [4] F. Cesarini and G. Soda. A dynamic hash method with signature. *ACM Transactions on Database Systems*, 16(2):309–337, 1991.

- [5] Gaston H. Gonnet and Per-Åke Larson. External hashing with limited internal storage. *J. Assoc. Comput. Mach.*, 35(1):161–184, 1988.
- [6] Peter Kjellberg and Torben U. Zahle. Cascade hashing. In Umeshwar Dayal, Gunter Schlageter, and Huat Seng Lim, editors, *Proceedings of 10th International Conference on Very Large Data Bases (VLDB)*, pages 481–492. Morgan Kaufmann, 1984.
- [7] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Publishing Co., Reading, Mass., 1973. Volume 3. Sorting and searching.
- [8] Murlidhar Koushik. Dynamic hashing with distributed overflow space: a file organization with good insertion performance. *Information system*, 18(5):299–318, 1993.
- [9] Per-Åke Larson. Linear hashing with partial expansions. In Canadian Information Processing Society, editor, *Proceedings of 6th International Conference on Very Large Data Bases (VLDB)*, pages 224–232. IEEE Comput. Soc. Press, 1980.
- [10] Per-Åke Larson. Linear hashing with overflow-handling by linear probing. *ACM Transactions on Database Systems*, 10(1):75–89, 1985.
- [11] Per-Åke Larson. Performance analysis of a single-file version of linear hashing. *The Computer Journal*, 28(3):319–329, 1985.
- [12] Per-Åke Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.
- [13] Witold Litwin. Linear hashing: A new tool for files and tables addressing. In *Proceedings of 6th International Conference on Very Large Data Bases (VLDB)*, pages 212–223. IEEE Comput. Soc. Press, 1980.
- [14] James K. Mullin. Tightly controlled linear hashing without separate overflow storage. *BIT (Nordisk tidskrift for informationsbehandling)*, 21(4):390–400, 1981.
- [15] K. Ramamohanarao and John W. Lloyd. Dynamic hashing schemes. *The Computer Journal*, 25(4):479–485, 1982.