Bulk Loading a Linear Hash File

Davood Rafiei and Cheng Hu

University of Alberta
{drafiei, chenghu}@cs.ualberta.ca

Abstract. We study the problem of bulk loading a linear hash file; the problem is that a *good* hash function is able to distribute records into random locations in the file; however, performing a random disk access for each record can be costly and this cost increases with the size of the file. We propose a bulk loading algorithm that can avoid random disk accesses by reducing multiple accesses to the same location into a single access and reordering the accesses such that the pages are accessed sequentially. Our analysis shows that our algorithm is near-optimal with a cost roughly equal to the cost of sorting the dataset, thus the algorithm can scale up to very large datasets. Our experiments show that our method can improve upon the Berkeley DB load utility, in terms of running time, by two orders of magnitude and the improvements scale up well with the size of the dataset.

1 Introduction

There are many scenarios in which data must be loaded into a database in large volumes at once. This is the case, for instance, when building and maintaining a data warehouse, replicating an existing data, building a mirror Internet site or importing data to a new DBMS. There has been work on bulk loading tree-based indexes (e.g. quadtree [6], R-tree [3] and UB-Tree [4]), loading into an object-oriented database (e.g. [15,2]) and resuming a long-duration load [10]. However, we are not aware of a bulk loading algorithm for a linear hash file. This may seem unnecessary, in particular, if both sequential and random disk accesses are charged a constant time; but given that a random access costs a seek time and half of a rotational delay more, a general rule of thumb is that one can get 500 times more bandwidth by going to a sequential access [5]. This seems to be consistent with our experimental findings.

There are a few complications with loading a linear hash file which need to be resolved. First, the file is dynamic and both the hash functions and the record locations change as more data is loaded. Second, the final structure of a hash file depends on factors such as data distribution, the split policy and the arrival order of the records. Third, without estimating a target hash layout, it is difficult to order the input based on the ordering of the buckets in the hash file.

Overview of Linear Hashing: Linear hashing is a dynamic hashing scheme that gracefully accommodates insertions and deletions by allowing the size of the hash file to grow and shrink [11]. Given a hash file with initially N_0 buckets and a hash function $h()$ that maps each key to a number, $h_0(key) = h(key)\ mod\ N_0$ is called a base hash function and $h_i(key) = h(key)\ mod\ 2^i N_0$ for $i > 0$ are called split functions where N_0 is typically chosen to be 1. Buckets are split when there is an overflow. Linear hashing does not necessarily split a bucket that overflows, but always performs splits in a deterministic linear order. Thus the records mapped to an overfilled bucket may be stored in an overflow bucket that is linked to the primary area bucket.

Loading a Linear Hash: Consider loading a linear hash file with $N_0 = 1$. the hash file initially has a single bucket and grows in generations to 2, 4, ..., 2^n buckets. In the 0^{th} generation, the hash file grows from a single bucket to two buckets. Every record of the old bucket with its least significant bit (referred to here as bit 0) set is moved to the new bucket. In the i^{th} generation, the hash has 2^i buckets and grows into 2^{i+1} buckets in a linear order. For each record key, the i^{th} bit of its hash value is examined and it is decided if the record must be moved to a newly-created bucket.

Paper Organization: Section 2 presents our bulk loading algorithm. In Section 3, we compare and contrast our methods to caching, which can be seen as an alternative to bulk loading. Section 4 presents and analyzes our experimental results. Finally, Section 5 reviews the related work and Section 6 concludes the paper and discusses possible extensions and future work.

2 Bulk Loading

Based on our analysis [13], the cost of loading can be reduced if we can reduce or eliminate random page accesses and record movements.

2.1 Straightforward Solutions and Problems

To avoid random disk accesses in loading a hash file, a general solution is to sort the records based on the addresses they are hashed to before loading. Unlike static hashing where each record is mapped to a fixed location, the address of a record in a linear hash file is not fixed and it changes as more records are inserted or deleted. Sorting the records based on the hash values is not also an option since there is not a single hash function.

An alternative is to estimate the number of generations a hash file is expected to go through, say r, and sort the records based on the function $h(key)\ mod\ 2^r$ of their key values. This solution can avoid random disk accesses if the hash file (after all the data is loaded) is in a state right at the beginning or the end of a generation, i.e. bucket 0 is the next bucket to split. Otherwise r, the number of bits used for sorting, is not a natural number. Clearly one can solve the problem using $\lceil r \rceil$ bits for addressing, for the cost of an underutilized hash file. But this can almost double the space that is really needed.

The design of a linear hash file (as discussed in the previous section) forces the records within each bucket to have a few least significant bits of their hash values the same. For instance, in the i^{th} generation, the hash values of the records in each bucket must all have their i least significant bits the same. It is clear that there is not a unique final layout that satisfies this constraint. The final layout, for instance, can vary with the order in which the records are inserted.

2.2 Input Ordering and Load Optimality

There are many different ways of ordering a given set of input records, and each ordering may result into a different hash file configuration. To reduce the number of possible hash layouts that we need to search for, we define some equivalent classes of layouts.

Definition 1. *Let $R(b)$ denote the set of records that are stored in either the primary bucket b or an overflow bucket linked to primary bucket b. Two linear hash layouts l_1 and l_2 are equivalent if (1) for every primary-area bucket b_1 in l_1, there is a primary-area bucket b_2 in l_2 such that $R(b_1) = R(b_2)$, and (2) for every primary-area bucket b_2 in l_2, there is a primary-area bucket b_1 in l_1 such that $R(b_1) = R(b_2)$.*

For the purpose of loading, two different configurations may be treated the same if both have the same space overheads and I/O costs. On the other hand, the construction costs of two equivalent layouts can be quite different. We develop a notion of optimality which to some degree characterizes these costs.

Definition 2. *Suppose a target hash file is fixed and has N primary-area buckets. An* optimal ordering *of the records is the one such that loading records in that order into the hash file involves no bucket splits nor record movements and no bucket is fetched after it is written.*

This notion of optimality does not provide us with an actual load algorithm but makes it clear that before a bucket is written, all records that belong to the bucket must be somehow grouped together. Furthermore, to avoid bucket splits and record movements, the final layout must be predicted before the data is actually loaded. Our bulk loading algorithm is presented next.

2.3 Our Algorithm

Suppose a final hash layout is fixed and it satisfies the user's expectation, for instance, in terms of the average number of I/Os per probe. Thus, we know the number of buckets in the hash file (the details of our estimation is discussed elsewhere [13]). For each record, r least significant bits of its hash value gives the address of the bucket where the record must be stored. As is shown in Alg. 1., before the split point is reached, $r = \lceil log_2 N \rceil$ bits. At the split point, the number of bits used for addressing is reduced by one. Since the input is sorted based on b least significant bits of the hash values in a reversed order, all records with the same $r_1, r_2 \leq b$ least significant bits are also grouped together. Hence the correctness of the algorithm follows. Furthermore, the input ordering satisfies our optimality criteria; after sorting, the algorithm does not perform any bucket splits or record movements and no bucket is fetched after it is written.

Algorithm 1. Bulk Loading a hash file

```
Estimate the number of primary buckets in the hash file and denote it with $N$;

$r_1 = \lfloor log_2 N \rfloor$; $r_2 = \lceil log_2 N \rceil$
Sort the records on $b \in [r_2, mb]$ least significant bits of their hash values in a reversed
order, where $mb$ is the maximum length of a hash value in bits;

Let $p = N - 2^{r_1}$ denote the next bucket that will split
$r = r_2$; $b = 0$; {current bucket that is being filled}
while there are more records do
  Get the next record $R$ with the hash value $H_R$;
  Let $h$ be the $r$ least significant bits of $H_R$;
  Reverse the order of the bits in $h$;
  if $h > b$ {the record belongs to the next bucket} then
    Write bucket $b$ to the hash file; $b++$;
    if $b \geq p$ {has reached the split point} then
      $r = r_1$;
    end if
  end if
  if bucket $b$ is not full then
    Insert $R$ into bucket b;
  else
    Write bucket $b$ to the hash file if it is not written;
    Insert $R$ into an overflow bucket;
  end if
end while
```

Lemma 1. *The total cost of Alg. 1. in terms of the number of I/Os is roughly the cost of sorting the input plus the cost of sequentially writing it.*

Proof See [13].

3 Caching vs. Data Partitioning

Caching the buckets of a hash file can reduce the number of I/Os and may be an alternative to bulk loading, if it can be done effectively. The effectiveness of caching mainly depends on the replacement policy that is chosen and the size of the available memory. When the memory size is limited, a "good" caching scheme must predict the probe sequence of the records and keep the buckets that are expected to be accessed in near future in memory. However, unless the data is ordered to match the ordering of the buckets in the hash file, the probe sequence is expected to be random and every bucket has pretty much the same chance of being probed. Therefore, it is not clear if any replacement policy alone can improve the performance of the loading. If we assume the unit of transfer between the disk and memory is a bucket, reducing the size of a bucket can reduce unused data transfers, thus improving the cache performance at load time (e.g. [1]). However, using a small bucket size can also increase the average access time for searches [9].

An alternative which turns out to be more promising (see Section 4.1) is to use the available memory for data reordering such that the probes to the same or adjacent buckets are grouped together. As in caching, the data is scanned once but partitioned into smaller chunks and each partition is buffered. Sorting each partition in the buffer reorders the records so that the records in the same partition which belong to the same or adjacent buckets are grouped together.

For testing and comparison, both caching and partitioning can be integrated into Berkeley DB which supports linear hashing through its so-called extended linear hash [14]. The database does use caching to boost its performance. When a hash file is built from scratch, all buckets are kept in memory as long as there is room. The size of the cache can be controlled manually. Berkeley DB provides a utility, called *db_load*, for loading but the utility does not do bulk loading. Our partition-based approach can be implemented within *db_load* (as shown in Alg. 2.) by allocating a buffer for data reordering. Alg. 2. is not a replacement for Alg. 1. but it is good for incremental updates, after an initial loading and when the hash file is not empty.

Algorithm 2. Modified *db_load* with data partitioning

```
Initialize the memory buffer
while there are more records do
    Read a record $R$ from the dataset and add it to the buffer
    if the buffer is full then
        Sort the records in the buffer based on their reversed hash values
        Insert all the records in the buffer into the hash table
        Clear the buffer
    end if
end while
```

Obviously, the size of the buffer can directly affect the loading performance. The larger the buffer, the more records will be grouped according to their positions in the hash table. If we assume the size of the available memory is limited, then the space must be somehow divided between a cache and a sort buffer. Our experiments in the next section shows that a sort buffer is more effective than a cache of the same size.

4 Experiments

We conducted experiments comparing our bulk loading to both the loading in Berkeley DB and our implementation of a naive loading. Our experiments were conducted on a set of URLs, extracted from a set of crawled pages in the Internet Archive [7]. Attached to each URL was a 64-bit unique fingerprint which was produced using Rabin's fingerprinting scheme [12]. We used as our keys the ascii character encoding of each fingerprint; this gave us a 16-bytes key for each record. Unless stated otherwise, we used a random 100-bytes charter string for data values. We also tried using URLs as our keys but the result was pretty much the same and were not reported. All our experiments were conducted on a Pentium 4 machine running Red Hat 9, with a speed of 3.0GHz, a memory of 2GB, and a striped array of three 7200 RPM IDE disks. We used the version 4.2.52 of Berkeley DB, the latest at the time of running our experiments.

For our experiments with Alg. 1., we set $b = mb$, except for the experiments reported at the end of Section 4.1; this made the sorting independent of the layout estimation and had a few advantages: (1) external sorting could be used, (2) the data read by our layout estimation could be piped to sorting, avoiding an additional scan of the data. There was not also much improvement in running time when the number of bits used for sorting was less. For instance, external sorting 180 million 130-byte records based on 16 bits took 85 minutes whereas a sort based on 64 bits took 87 minutes. Our timings reported for Alg. 1. include the times for both sorting and layout estimation. For sorting, the Linux sort command was used.

4.1 Performance Comparison to Loading in Berkeley DB

As a baseline comparison, we used the native *db_load* utility in Berkeley DB and compared its performance to that of our bulk loading. *db_load* had a few parameters that could be set at load time including the fill factor (h_ffactor) and the number of records (*h_nelem*). In particular, when *h_nelem* was set, *db_load* did a layout estimation and built the entire empty hash table in advance. We played with these parameters, trying to find the best possible settings. In our experiments, however, we did not notice any performance improvements over default settings, except in those cases where the input followed a specific ordering as discussed at the end of this Section. Otherwise, the performance even deteriorated when the parameters were explicitly set. Therefore, unless stated otherwise, we used the default settings of the *db_load* utility.

Scalability with the size of the dataset. To test the scalability of our algorithms and to compare caching (in Berkeley DB) with our partitioning, we varied the size of the dataset from 1 million to 20 million records and measured the running time for Alg. 1., Alg. 2. and the native *db_load*. The size of the sort buffer in Alg. 2 was set to 300MB (our next experiment shows how the buffer size can affect the load performance). If we included the 1MB I/O cache which was automatically allocated by Berkeley DB, the total memory allocated to Alg. 2 was 301MB. To make a fair comparison, we also set the I/O cache of the native *db_load* utility to 301MB. All other parameters were set to their default values in Berkeley DB.

The result of the experiment is shown in Fig. 1-a. When the dataset is small (less than 5 million records), all three methods perform very well and their performances are comparable. This is because both the I/O cache of the native *db_load* and the sort buffer of Alg. 2 are large enough to hold a major fraction of data. When the dataset size is 5 million records (i.e. 590MB), half of the input data cannot fit in the sort buffer of Alg. 2 or the I/O cache of the native *db_load* utility, and Alg. 2 improves upon the native *db_load* by a factor of 1.5. When the dataset contains more than 10 million records, our experiment shows that Alg. 2 outperforms the native *db_load* utility by at least a factor of 3. The performance of our bulk loading algorithm is better than the other two approaches. It takes only 10 minutes and 23 seconds to load the dataset with 20 million records while native *db_load* utility in Berkeley DB requires 1682 minutes and 1 seconds.

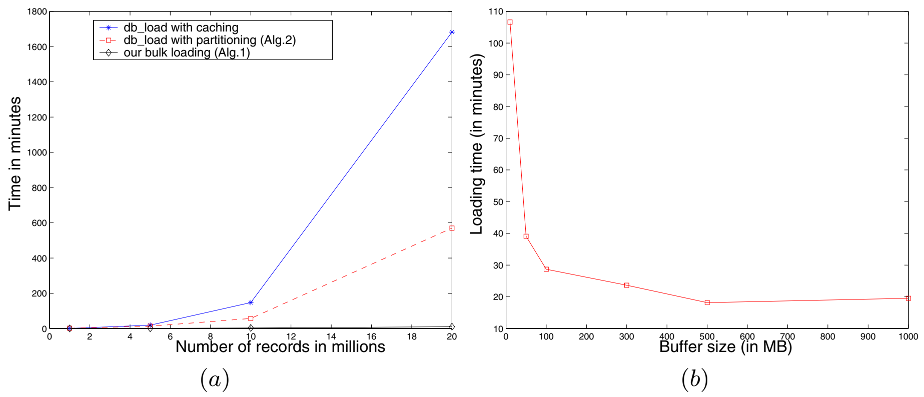

Fig. 1. Running time varying (a) the number of records, (b) the buffer size

Buffer size. As discussed in the previous section, when the dataset cannot be fully loaded into memory, the sort buffer is always more effective than an I/O cache of the same size. In another experiment to measure the effect of the sort buffer size on the performance, we fixed the size of the dataset to 10 million records and varied the sort buffer size in Alg. 2 from 100MB to 1GB. Each record contained a 16-bytes key and a 50-bytes data field. The default I/O cache size of *db_load* was 1MB. The result in Fig. 1-b shows that allocating a modest size buffer for sorting (in this case less than 50MB) sharply reduces the running time. Clearly allocating more buffer helps but we don't see a significant drop in running time. This is a good indication that our partitioning can be integrated into other applications with only a small buffer overhead.

Sorting data in advance. In an attempt to measure the effect of input ordering alone (without a layout estimation), we sorted the records based on i least significant bits of their hash values with i varied from 0 to 32, where 32 was the length of a hash value in bits. As is shown in Fig. 2-a for 10 million records of our URL dataset, the loading time was the worst when data was not sorted or the sorting was done on the whole hash value. Increasing i from 0 toward 32 reduced the running time until i reached a point (here called an optimal point) after which the running time started going up. The optimal point was not fixed; it varied with both the size of the dataset and the distribution of the hash values. However, if we reversed the bit positions before sorting, increasing i from

0 toward 32 reduced the running time until i reached its optimal point after which the running time almost stayed the same[1]. Clearly sorting improves the performance when data is sorted either on the reversed hash values or on the original order but using an optimal number of bits.

We could not do our layout estimation in Berkeley DB but could pass the number of records and let Berkeley DB do the estimation. In another experiment we sorted the data and also passed the number of records as a parameter to the load utility. Fig. 2-b shows the loading time for the same 10 million record dataset when the number of records is passed as a parameter and the number of bits used for sorting, i, is varied from 1 to 32. A layout estimation alone (i.e. when $i = 0$) did not improve the loading time; this was consistent with our experiments reported earlier in this section. Comparing the two graphs in Fig. 2 leads to the conclusion that the best performance is obtained when sorting is combined with a layout estimation (here the layout estimation is done in Berkeley DB).

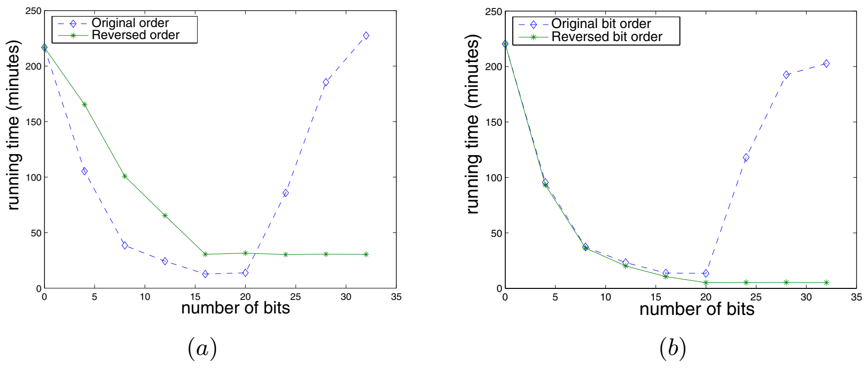

Fig. 2. Loading sorted data using *db_load* (a) without the number of records set, (b) with the number of records set

4.2 Performance Comparison to Naive Loading

We could not run Berkeley DB for datasets larger than 20 million records as it was either hanging up or taking too long [2]. Therefore we decided to implement our own loading, here called *naive loading*, which as in Berkeley DB inserted one record at a time but did not have the Berkeley DB overheads due to the implementation of ACID properties. To compare the performance of this naive loading to that of our bulk loading (Alg. 1.), we varied the size of the dataset from 1 million to 50 million records and measured the loading time. We couldn't run the naive loading for larger datasets; it was taking already more than 55 hours to run it with 50 million records. The full result of the comparison could not be

[1] Increasing i may slightly increase the time for sorting, but this increase (as discussed at the beginning of this section) is negligible.

[2] For instance, loading 20 million records took over 26 hours (see Fig. 1-a).

presented due to space limitations, but loading 10 million records, for instance, using our bulk loading algorithm took 3 minutes and 16 seconds whereas it took 129 minutes and 55 seconds to load the same dataset using the naive algorithm. For 50 million records, using our bulk loading algorithm took 27 minutes and 4 seconds whereas naive algorithm needed 3333 minutes and 18 seconds. Generally speaking, our bulk loading algorithm outperforms the naive loading by two orders of magnitude, and its performance even gets better for larger datasets.

5 Related Work

Closely related to our bulk loading is the incremental data organization of Jagadish et al. [8] which delays the insertions into a hash file. They collect the records in piles and merge them with the main hash only after enough records are collected. Data in each pile is organized as a hash index and each bucket of the index has a block in memory. This idea of lazy insert is similar to our Alg. 2.. A difference is that we use sorting, thus the records that are mapped to the same location in the hash file are all adjacent. This may provide a slight benefit at the load time. Our Alg. 1. is different and should be more efficient. The entire data is sorted in advance using external sorting which is both fast and scalable to large datasets; it is also shown that the total cost of the algorithm is roughly equal to the cost of sorting. On a dataset with 20 million records, Alg. 1. is 50 times faster than our partition-based algorithm (Alg. 2.) which is comparable to a lazy insertion of Jagadish et al. [8].

To the best of our knowledge, hash indexes are not currently supported in DB2, Sybase and Informix; this may change as these databases provide more support for text and other non-traditional data. Hash indexes are supported in Microsoft SQL Server, Oracle (in the form of hash clusters), PostgreSQL and Berkeley DB (as discussed earlier), but we are not aware of any bulk loading algorithm for these indexes.

6 Conclusions

Hash-based indexes are quite attractive for searching large data collections, because of their low cost complexity, however the initial time for loading is a major factor in the adoption of a hash-based index in the first place. Our work, motivated by our attempt to load a snapshot of the Web into a linear hash file, presents a few algorithms for efficiently loading a large dataset into a linear hash file. Our analysis of these algorithms and our experiments show that our algorithms are near-optimal, can scale up for large datasets and can reduce the loading time by two orders of magnitude.

Acknowledgments

The authors would like to thank Margo Seltzer and Keith Bostic for answering many of our questions about Berkeley DB and comments on our earlier draft and Paul Larson for the discussions. This work is supported by Natural Sciences and Engineering Research Council of Canada.

References

1. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: Proceedings of the VLDB Conference, Rome, Italy (2001) 169–180
2. Amer-Yahia, S., Cluet, S.: A declarative approach to optimize bulk loading into databases. ACM Transactions on Database Systems **29**(2) (2004) 233–281
3. Böhm, C., Kriegel, H.: Efficient bulk loading of large high-dimensional indexes. In: International Conference on Data Warehousing and Knowledge Discovery. (1999) 251–260
4. Fenk, R., Kawakami, A., Markl, V., Bayer, R., Osaki, S.: Bulk loading a data warehouse built upon a ub-tree. In: Proceedings of of IDEAS Conference, Yokohoma, Japan (2000) 179–187
5. Gray, J.: A conversation with Jim Gray. ACM Queue **1**(4) (2003)
6. Hjaltason, G.R., Samet, H., Sussmann, Y.J.: Speeding up bulk-loading of quadtrees. In: Proceedings of the International ACM Workshop on Advances in Geographic Information Systems, Las Vegas (1997) 50–53
7. Internet Archive: (http://www.archive.org)
8. Jagadish, H.V., Narayan, P.P.S., Seshadri, S., Sudarshan, S., Kanneganti, R.: Incremental organization for data recording and warehousing. In: Proc. of the VLDB Conference, Athens (1997) 16–25
9. Knuth, D.: The Art of Computer Programming: Vol III, Sorting and Searching. Volume 3rd ed. Addison Wesley (1998)
10. Labio, W., Wiener, J.L., Garcia-Molina, H., Gorelik, V.: Efficient resumption of interrupted warehouse loads. In: Proc. of the SIGMOD Conference, Dallas (2000) 46–57
11. Larson, P.: Dynamic hash tables. Communications of the ACM **31**(4) (1988) 446–457
12. Rabin, M.O.: Fingerprinting by random polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University (1981)
13. Rafiei, D., Hu, C.: Bulk loading a linear hash file: extended version. (under preparation)
14. Seltzer, M., Yigit, O.: A new hashing package for unix. In: USENIX, Dallas (1991) 173–184
15. Wiener, J.L., Naughton, J.F.: OODB bulk loading revisited: The partitioned-list approach. In: Proceedings of the VLDB Conference, Zurich, Switzerland (1995) 30–41