University of Alberta

Bulk Loading a Linear Hash Index

by

Cheng Hu 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2005

Abstract

In this thesis, we study the problem of bulk loading a linear hash index and address some of the challenges that arise when loading a large data set. 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 one and reordering the accesses such that the pages are accessed sequentially. Our analysis shows that our algorithm is optimal with a cost roughly equal to the cost of sorting the data set, thus the algorithm can scale up to very large data sets. We integrate our algorithm into Berkeley DB and run experiments to compare the performance of our method to the native load utility in Berkeley DB. The result shows that our method can improve upon the Berkeley DB load utility, in terms of running time, by two orders of magnitude.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

There are many applications 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 Database Management System (DBMS). There has been work on bulk loading tree-based indexes (e.g. quadtree [8], R-tree [3] and UB-Tree [6]), loading into an object-oriented database (e.g. [16, 18, 19]), loading into a multidimensional index structures (e.g. [9]) and resuming a long-duration load [11]. However, we are not aware of a bulk loading algorithm for a hash index [1]. 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 on average 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 [7]. This seems to be consistent with our experimental findings.

Efficiently loading data into a hash-based index is useful in many applications which require direct access to data through equality queries but no range searches. When the data set is quite large, the load process may take

[1] One of our algorithms has similarities to the incremental data organization of Jagadish [10] but is different; see Sec. 6 for more details.

hours or even days, as some reported in our experiments. The problem to be addressed in this thesis is to optimize the time to load by both minimizing the number of redundant disk I/Os and reducing or eliminating random disk accesses. For linear hashing, in particular, redundant I/Os are mainly due to bucket splits and record movements. Furthermore, since the location of each record in the hash file is determined by a hash function which is expected to produce a random number, loading the records into the hash file can cause a large number of random disk accesses; this obviously reduces the efficiency of the loading process.

Our proposed solution is to predict the final structure of the hash file before the data is actually loaded. If we can order the records based on their estimated locations in the file, then we can do the loading within one sequential scan. However, there is not a unique final layout for a given data set; often the final structure varies depending on the order of the insertions and the split policy that is used. Our objective is to find a balance between the access cost and the storage overhead. To this end, we develop a user-tunable cost function that guides the load toward an optimal layout. Once an optimal layout is fixed, we accordingly reorder the records to ensure that the records that belong to the same bucket are loaded together. With this strategy, the bucket splits and record movements are avoided and the disk accesses all become sequential.

We run experiments with real data and compare the timings of our bulk loading algorithm to the native *db_load* utility in Berkeley DB [5], an open-source embedded database library. Our experiment with loading a data set of 30 million 100-byte records, for instance, shows that our bulk loading algorithm runs 150 times faster than the *db_load* utility using a Pentium 4 3.0G, 2GB memory machine running on Red Hat 9.0.

Based on this and other experiments reported in Chapter 5, we should expect for our load algorithms a performance improvement, in terms of running

time, of roughly two orders of magnitude. Without loss of generality, we will base our algorithms and discussions on linear hashing [12, 14], mainly because of its efficient dynamic structure and also its frequent use in practice [5]. It shouldn't be hard to extend our algorithms to other hash-based indexes because of the similarities between these indexes.

This thesis is organized as follows: Chapter 1 provides some background on linear hashing and an I/O analysis of loading a linear hash file. Chapter 2 presents our bulk loading algorithm. Finding an optimal setting of our bulk loading parameters is discussed in Chapter 3. In Chapter 4, we discuss the details of partially integrating bulk loading into the *db_load* utility from Berkeley DB. Chapter 5 presents and analyzes our experimental results. Finally, Chapter 6 reviews the related work and Chapter 7 concludes the thesis and discusses possible extensions and future work.

1.2 Hash-based Index

The basic idea of hash-based index is to use a hashing function, which maps a record key into an address space, to find the page on which a desired data entry belongs. The most common techniques for hashing are static hashing and dynamic hashing.

The static hashing scheme has a fixed number of hash buckets. The pages containing the data are called hash buckets. A hash layout consists of buckets 0 through $N-1$, with one primary page per bucket at the beginning. To search for a data entry, a hash function h is used to identify the bucket to which it belongs. When a record is inserted, the hash function is used to identify the correct hash bucket and then put the record into this bucket. If there is no space for this record in the bucket, we allocate a new overflow page, insert the record into this overflow page, and add the page to the overflow chain of the bucket. When a record is deleted, we also use the hashing function to

identify the correct bucket, find the data entry by searching in the bucket and then remove it. The main problem of static hashing is that the number of buckets is fixed. Therefore, if a hash file grows a lot, long overflow chains may be generated causing poor performance. If a file shrinks greatly, many hash buckets may become empty and a lot of space is wasted.

An alternative is to use a dynamic hashing technique such as extendible or linear hashing. In static hashing, when we insert a record into a full bucket, we need to add an overflow bucket. If we don't want to add an overflow bucket, we have to reconstruct the hash index by doubling the number of buckets and redistributing all the records in the hash table into the new address space. This solution suffers from too may redundant I/Os. All the records in the entire file have to be read and half of the records are written to the new buckets. Therefore, twice as many pages in total have to be written.

The idea of dynamic hashing is that instead of splitting all of the hash buckets and doubling the number of buckets, only the bucket that overflowed are split and the splits are performed in a deterministic order. A bucket split is to split the bucket by allocating a new bucket and redistributing the records across the old bucket and its split image. The associated hash function must change as the table grows. Some schemes may shrink the table to save space when items are deleted.

Extendible hashing is a well-known dynamic hashing technique. The main elements of an extendible hashing structure are a directory of pointers, which points to buckets that contain the records. The size of the address space can be doubled by doubling just the size of the directory of pointers but splitting only the bucket that overflowed. The directory always has a size of a power of 2. The number of bits for addressing in the directory is called the global depth. A local depth is also maintained for each bucket. At the beginning, all local depths are equal to the global depth. To search a key in an extendible

hashing index, it always find the appropriate entry in the directory based on its hash value, and follows the pointer to the bucket contains the record. A record is inserted into the bucket which it belongs using the same method. A bucket split leads to an increase in the local depth and the split image is assigned the same local depth. If the local depth becomes greater than the global depth, a directory doubling occurs and the global depth itself is also increased by 1.

If the directory fits in memory, an equality search only requires a single disk access, which is the same as static hashing (in the absence of overflow pages). Otherwise, two disk accesses are needed. However, chances are high that the directory will fit in memory and the performance of extendible hashing is the same as for static hashing.

Linear hashing is another dynamic hashing technique. Compared to extendible hashing, the directory structure can be avoided by allocating primary hash buckets consecutively. The details of this hashing technique is discussed in the next section. In this thesis, we focus on the bulk loading method on linear hashing. However, it shouldn't be hard to extend our bulk loading algorithm to hashing techniques other than linear hashing because of the similarities between these indexes.

1.3 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 [12]. It is known to exhibit a near-optimal performance in terms of both the access cost and the storage overhead [12]. Given a hash file with initially N buckets and a hash function $h()$ that maps each key to a number, $h_0(key) = h(key) \bmod N$ is called a base hash function and $h_i(key) = h(key) \bmod 2^i * N$

for $i > 0$ are called split functions where

$$h_i(key) \rightarrow \{ 0, 1, \ldots, 2^i * (N-1) \}$$

and

$$h_i(key) = h_{i-1}(key)$$

$$or \quad h_i(key) = h_{i-1}(key) + 2^{i-1} * N.$$

Suppose we want to insert a record with $h_0(key) = 0$ into bucket 0 and this bucket overflows. Let bucket 0 be the first bucket that overflows in the file. The split function $h_1(key)$ is then used, and all records with $h_1(key) = h_0(key)$ are kept in bucket 0 and all records with $h_1(key) = h_0(key) + N$ are moved to the new bucket N.

Linear hashing does not necessarily split a bucket that overflows, but always performs splits in a deterministic order. The buckets are split in a linear order, starting from bucket 0 and following in buckets 1, 2, ..., $N-1$. A split usually occurs when there is an overflow. If the bucket that is split is not the bucket that overflows, then the overflow record may be stored in an overflow area and is chained to its home location bucket. The number of overflow buckets is usually kept small and a search for a specific record in the file is expected to take one or at most two disk accesses. The situation to trigger a split is also very flexible. We can split whenever a new overflow page is added or impose additional conditions such as space utilization in the hash table.

1.4 Loading: An I/O Analysis

Given a set of records to be inserted to a hash file, a bit-randomizing hash function is used to convert each record key into a k-bit hash value. This bit-randomizing property of the hash function is important to obtain radically different hash values for nearly identical keys [17]. The hash table initially has a single bucket and grows in generations to 2, 4, ..., 2^n buckets.

A dynamic hash file begins with a single bucket and grows in so-called generations. 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 the new bucket. In the i^{th} generation, the hash file has 2^i buckets which split 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 the newly-created bucket.

There are two major factors that can affect the load performance: reading and writing records into the hash table, and moving records from an old hash bucket to a newly-generated bucket. For a data set with n records, let P be the average record size in pages (usually, $P < 1$). The data set itself is $n * P$ pages. Reading these n records requires $n*P$ page accesses. Since the location of each record in the hash table is random due to the bit-randomizing hash function — in the worst case, writing a record into the hash table requires one page access. Therefore, the total cost of reading and writing is

$$n * P \ + \ n$$

I/Os. The cost of writing the records in practice can be slightly less due to the buffering, in particular, if a large fraction of the buckets are found in the buffer. However, in general the hash probes (i.e. searches for the locations of the new records) are randomly distributed, and almost every probe is likely to find the corresponding bucket not in the buffer.

The load performance is also affected by the number of record movements. Given a well-designed bit-randomizing hash function, it is safe to assume that for any bit position i, half of the records have their i^{th} bit set. Therefore, within the i^{th} generation, half of the records in the former 2^i buckets are expected to move to the newly-generated buckets. Let m denote the average number of records in a hash bucket. We need n/m buckets for the hash file. The total

number of record movements is

$$\begin{aligned} & (2^0 + 2^1 + 2^2 + ... + 2^{[(\log_2 \frac{n}{m})-1]}) * \frac{m}{2} \\ = & (2^{\log_2 \frac{n}{m}} - 1) * \frac{m}{2} \\ = & (\frac{n}{m} - 1) * \frac{m}{2} \\ \approx & \frac{n}{2}. \end{aligned}$$

Therefore, half of the records are initially loaded into buckets other than their home buckets. There is a cost associated to move these records into their home locations. To slightly simplify our analysis, suppose reading or writing a hash bucket requires one disk I/O. Without considering the effect of buffering, within the 0^{th} generation, we need to read one bucket, redistribute some of its records to a new bucket and write back two buckets. Similarly, in the i^{th} generation, we need to read 2^i buckets and write back 2^{i+1} buckets. Since we need n/m buckets for the hash file, constructing this hash file requires

$$\begin{aligned} & 2^0 + 2^1 + 2^2 + ... + 2^{[(\log_2 \frac{n}{m})-1]} \\ = & 2^{\log_2 \frac{n}{m}} - 1 \\ = & \frac{n}{m} - 1 \end{aligned}$$

disk reads and

$$\begin{aligned} & 2^1 + 2^2 + 2^3 + ... + 2^{(\log_2 \frac{n}{m})} \\ = & 2^{[(\log_2 \frac{n}{m})+1]} - 2 \\ = & \frac{2n}{m} - 2 \end{aligned}$$

disk writes. Without taking into account the buffering effects, there are

$$\frac{3n}{m} - 3$$

I/Os due to bucket splits. Thus the total cost of loading n records is

$$n * P + n + \frac{3n}{m} - 3$$

I/Os. To have an idea of this cost, loading 20 million 100-bytes records takes up to 16 hours in our experiments. This is clearly not acceptable for such a data set using a relatively modern hardware in 2005.

In another aspect, if the records are uniformly distributed in the address space, linear hashing has a lower average cost for equality queries than the extendible hashing because the directory level is eliminated. However, for skewed distributions, linear hashing could result in many empty or nearly empty buckets, leading to poor performance. The reason is linear hashing does not necessarily split a bucket that overflows. Efficiently loading a skewed data set into a linear hash index with a performance comparable to that of loading random data is another challenge.

Chapter 2

Bulk Loading

Our analysis in Section 1.4 reveals that the cost of loading can be reduced if we can take the following actions:

- Reduce the number of random page accesses and/or replace them with sequential page accesses when writing the records into the hash file. Since the final location of a record in the hash file is determined by a hash function which is expected to produce random numbers, writing the records in their original order of arrival can generate a random disk access for each record. This is quite costly and must be avoided.

- Reduce or eliminate bucket splits and record movements. If we can predict the final hash layout and store each record in its predictive final destination bucket, bucket splits and record movements can be avoided.

To avoid random disk accesses, a solution is to sort the records according to the order they are expected to sit in the final hash file. 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: in the i^{th} generation, the hash values of the records in each bucket must all have their i least significant bits the same. There is not a unique final layout that satisfies this constraint (as discussed in the next section). The order of

the records in one bucket of the final layout, for instance, can vary with the order in which the records are inserted.

2.1 Multiple Hash Layouts

There are many different ways of ordering a given set of records, and as a result there are many possible configurations of a hash file. Two different configurations may be treated the same if both have the same space overheads and I/O costs. To reduce the number of possible hash layouts, we define some equivalent classes.

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 $R(b_2) = R(b_1)$.

Based on this definition, the two hash layouts shown in Figure 2.1, for instance, are equivalent. Naturally, the ordering of the records within primary-area and overflow-area buckets and the ordering of the buckets in two equivalent layouts can be different. The reason is that even using the same bit-randomizing hash function, there may be different ways to map records into the address space according to their hash values.

Lemma 1. *Suppose all records are of a fixed length and the overflow records are stored in overflow-area buckets and are chained to some primary-area buckets. For any pairs of equivalent layouts, the following holds:*

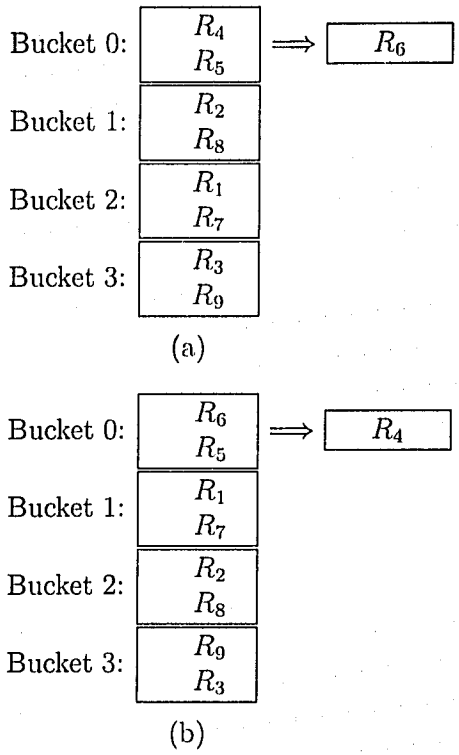

Figure 2.1: Two equivalent layouts

1. *the space requirements, in terms of the number of primary-area and overflow-area buckets, of both layouts is the same,*

2. *the average number of I/Os required for a probe in both layouts is the same if every record has the same chance of being probed.*

The proof can be derived from our definition of equivalent hash layouts. For two equivalent hash layouts with fixed length records, only the ordering of the records within buckets and the ordering of the buckets can be different. These order differences can not affect the space requirements and average I/Os per probe. For records of variable lengths or when overflow records are handled using a different strategy than the one in Lemma 1, the storage overhead and the I/O cost of two equivalent layouts are still expected to be close (if not the

same). On the other hand, the construction costs of two equivalent layouts can be quite different. Consider the two layouts in Figure 2.1, and suppose the hash values of the records are those shown in Table 2.1.

Records	Hash values
R_1	(1 0 1 0)
R_2	(0 1 0 1)
R_3	(1 1 1 1)
R_4	(1 1 0 0)
R_5	(0 1 0 0)
R_6	(1 0 0 0)
R_7	(0 1 1 0)
R_8	(1 1 0 1)
R_9	(1 0 1 1)

Table 2.1: Nine records with their hash values.

The hash file in Figure 2.1(a) is the result of inserting these records in the given order into a linear hash file. There are 3 bucket splits and 3 records movements. If we fix the size of the buffer to one page (for our illustration purpose), then the first three buckets must be fetched more than once. The hash file in Figure 2.1(b) is the result of sorting the records based on the two least significant bits of their hash values (after reversing the positions of the bits) and filling the buckets sequentially. There is no bucket splits nor records movements, and each bucket is fetched only once.

2.2 Optimal Algorithm

Some of the major costs in loading a hash index are associated with the bucket splits, record movements and fetching a bucket more than once. We develop a notion of optimality of a load algorithm to avoid these costs.

Definition 2. A load algorithm is s-optimal if it can find and use an ordering of the records such that loading the records in that order does not involve any bucket splits or record movements and it does not fetch a bucket 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 known before the data is actually loaded.

Our bulk loading algorithm estimates both the distribution of the records and a layout that best fits this distribution. Details of our estimations are discussed in Section 3. Our algorithm also sorts the records such that the records that belong to the same bucket are grouped together. An overview of our algorithm is shown in Figure 2.2.

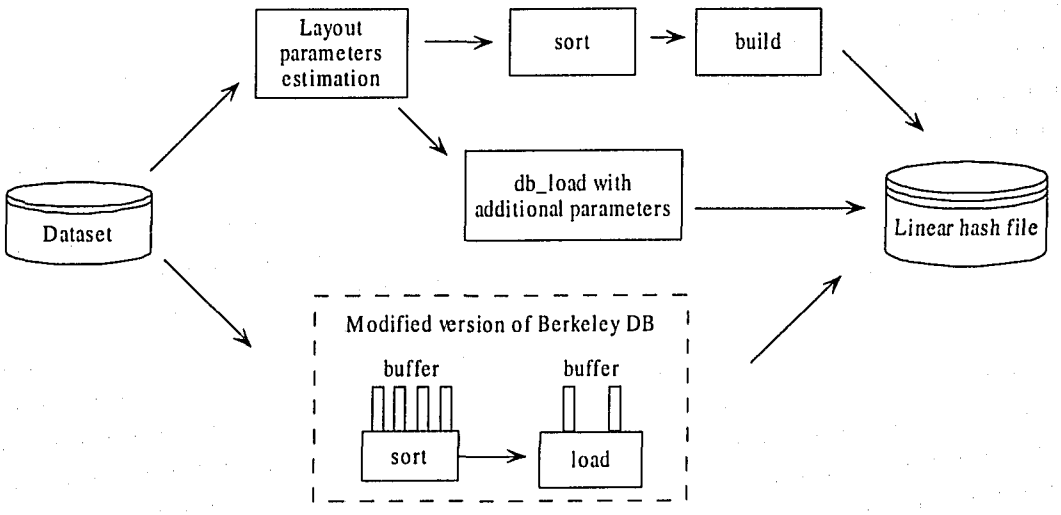

Figure 2.2: Overview of our bulk loading

Figure 2.2 provides two different methods of bulk loading records into a linear hash index. The fist method is our full algorithm (shown on top), which is used when we load a data set into an empty hash table. We predict the hash layout before loading, reorder the records in the data set and then write the records into the hash table. The hash table of this method is a static hash

table. The second method (shown at the bottom) can be used with both empty and non-empty hash files, and is discussed in Chapter 4. This method uses a buffer with a dynamic hash table. If we want to insert records incrementally into the hash index when they are received, the second method is preferred. In this thesis, we implement the first method independently, but integrate our second method into Berkeley DB, which is an open source database library. The details of these two methods are discussed in the following sections.

2.3 Our Algorithm

Algorithm 1 presents the details of our bulk loading method. Suppose the optimal number of buckets N in the final hash layout is somehow estimated; the details of our estimation is discussed in the next chapter. For records in the data set, the number of bits of the hash values for addressing can be calculated as $\lfloor log_2 N \rfloor$ or $\lceil log_2 N \rceil$. We reorder the records in the data set according to their positions in the predictive final hash file, i.e. the records are sorted based on the right number of bits for of their hash values in a reversed order. Finally, we insert these records into the hash table. There is no buckets splits or records movements in the last step because both the number of bits for addressing and the number of hash buckets in the final hash file are known.

In Algorithm 1, for each record, r least significant bits of its hash value gives the address of the bucket where the record must be stored. Before the split point is reached, the algorithm uses $\lceil log_2 N \rceil$ bits for addressing. At the split point, the number of bits used for addressing is reduced by one to indicate that the buckets after that point are not split. When $log_2 N$ is an integer, the resulting hash file has the state of being at the beginning (or the end) of a generation.

Lemma 2. *Algorithm 1 is s-optimal.*

Algorithm 1 Bulk Loading a hash index

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 r_2 least significant bits of their hash values in a reversed order;

Let $p = N - 2^{r_1}$ denote the next bucket that will split

$b = 0$; {current bucket that is being filled}
$r = r_2$;
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;
    if there is an overflow bucket with enough room **then**
      insert R into an overflow bucket;
    else
      insert R into a new overflow bucket;
      link the overflow bucket to bucket b or the other overflow bucket (if any);
      Write the overflow bucket when it gets full;
    end if
  end if
end while

This is because there are no buckets splits and records movements and buckets are not accessed in the algorithm after they are written. The purpose of hashing is to distribute records randomly into the address space. We choose the last few bits as the sorting bits because these bits are less significant compared to the first few bits of the hash values. For many hash functions, the fist few bits of the hash values they generated are the same.

Estimating the number of primary buckets can be done within on scan and while the records are being read for sorting, hence it involves no additional I/Os. Further to sorting, the data is read once and written once and both are done sequentially. If we pipe the result of the sort to our loading, there is no additional reading. Thus the total cost of the algorithm is the cost of sorting the records plus the cost of sequentially writing them.

Chapter 3

Layout Parameter Estimation

3.1 Layout Parameter Estimation based on a user specified I/O

An important part of our algorithm is the prediction of a "good" hash layout before data is actually loaded. This prediction is not generally easy for a dynamic hash file since the final layout depends on both the distribution of the data and the order in which the records are inserted. Our goal in this section is to find a hash layout with a user-specified average I/O for retrieving a record.

A good hash layout should have a low average access cost while keeping the storage overhead small. Clearly, improving the access cost involves increasing the storage overhead and vice versa, because a compact hash file with a high record density has a greater I/O cost per hash probe. To find a good trade-off, we use a user-tunable I/O cost and seek a layout that optimizes this function.

Our bulk loading algorithm is independent of the input ordering; we exploit the fact that the records in every bucket of a final layout has their few least significant bits the same. We also look for the best layout (for some given parameters) that is equivalent to a final layout, but there may not be any ordering of the input that produces that particular layout via incremental insertions.

Let L denote the capacity of a bucket, i.e. the maximum number of records that can be stored in a bucket. For variable length records, L can be calculated as the ratio between the bucket size in any units and the average record size again in the same units. Without loss of generality, suppose reading or writing a bucket involves one I/O, and that to access a record in a bucket, we need to retrieve the whole bucket. Therefore, retrieving a record in a primary hash bucket requires one I/O, and retrieving a record in an overflow bucket requires more than one I/O.

We do our estimation for the following two cases: (1) the hash function distributes the records uniformly at random in the address space, and (2) the hash function is skewed toward a few addresses.

3.1.1 Layout Estimation for Randomly Distributed Data

Let N denote the number of primary buckets in the hash file. We want to find a value of N that optimizes the storage usage and still guarantees a hash layout that has a user specified average cost for retrieving a record. Suppose the hash function is chosen such that it randomly maps each record key into a k bit hash value. When the records are randomly distributed into the address space, the probability that an arbitrary bucket in the final layout has x records mapped to it can be predicted using the following binomial distribution:

$$C_x^n \left(\frac{1}{N}\right)^x \left(1 - \frac{1}{N}\right)^{(n-x)} \tag{3.1}$$

where n is the total number of records and $C_x^n = n!/(x!(n-x)!)$. Computing the expression in Equation 3.1 in practice is not easy. Instead, a good approximation of this function which is easier to compute for large values of n and N can be given using the following Poisson distribution:

$$P(x) = e^{-\lambda}\frac{\lambda^x}{x!} \tag{3.2}$$

where $\lambda = n/N$. The parameter λ can be treated as the fill factor of the hash file, which is the average number of records in a hash bucket. We can use Equation 3.2 to estimate the distribution of records in the final hash layout. The expected number of buckets with x records is $N * P(x)$.

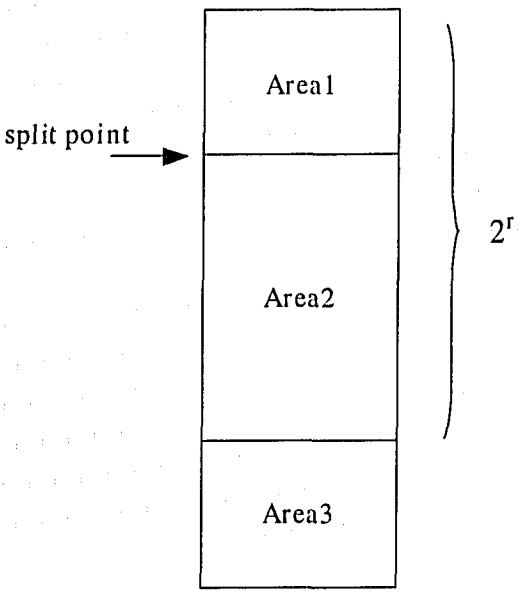

Figure 3.1: A linear hash file

Equation 3.2 is based on the assumption that the records are distributed randomly in the "entire" address space. However, this assumption often does not hold for linear hashing. Take the linear hash file in Figure 3.1 as an example, where the file consists of three regions with different record densities. Area 1 includes the buckets which are split; suppose this region has N_1 buckets. Area 2 includes the buckets which are not yet split in the current generation; suppose this region has N_2 buckets. Area 3 also has N_1 buckets since every bucket in Area 1 has its buddy bucket in Area 3. There are two hash functions; r bits are used for addressing a record in areas 1 and 3, whereas $r-1$ bits are used to address a record in Area 2.

Suppose there are n records and N hash buckets in the entire hash table.

We have the following constraints:

$$N_1 + N_2 + N_1 = N$$

$$N_1 + N_2 = 2^r = 2^{\lfloor \log_2 N \rfloor}$$

From these equations, N_1 and N_2 can be rewritten in terms of N as follows:

$$N_1 = N - 2^{\lfloor \log_2 N \rfloor} \tag{3.3}$$

$$N_2 = 2^{(\lfloor \log_2 N \rfloor + 1)} - N \tag{3.4}$$

The fill factor λ_1 or the average number of records in a hash bucket of Area 1 and Area 3 is

$$\lambda_1 = \frac{n}{2 * (N_1 + N_2)} = \frac{n}{2^{\lfloor \log_2 N \rfloor + 1}} \tag{3.5}$$

Plugging λ_1 in Equation 3.2, we can find the probability that a bucket in Area 1 and Area 3 has x records. The average cost to retrieve a record is the ratio of the number of I/Os to retrieve all records and the number of records. Here, any possible buffering effect is not accounted for. Let $Tb(i)$ denote the number of I/Os to retrieve all i records mapped to a primary-area bucket:

$$Tb(i) = \sum_{j=0}^{\lfloor i/L \rfloor} (i - j * L) \tag{3.6}$$

For example, with L, the capacity of a hash bucket, set to 10, $Tb(15)$ is 20; i.e., 10 I/Os for retrieving the records that are physically stored in the primary-area bucket and 2*5 I/Os for the records that are stored in an overflow-area bucket. Hence, the total number of I/Os to retrieve all records in Area 1 and Area 3 can be computed as:

$$T_1(N) = 2 * \sum_{i=1}^{\infty} N_1 * P(i) * Tb(i) \tag{3.7}$$

Similarly, the fill factor of Area 2 is:

$$\lambda_2 = \frac{n}{N_1 + N_2} = \frac{n}{2^{\lfloor \log_2 N \rfloor}} \tag{3.8}$$

and the total number of I/Os to retrieve all records in Area 2 is:

$$T_2(N) = \sum_{i=1}^{\infty} N_2 * P(i) * Tb(i) \tag{3.9}$$

Thus, the average number of I/Os to retrieve a record from the whole hash table is:

$$f(N) = \frac{T_1(N) + T_2(N)}{n} \tag{3.10}$$

Lemma 3. *The average number of I/Os, f, is a monotonically non-increasing function of N; N is also a monotonically non-increasing function of f.*

Lemma 3 can be derived from the fact that when N increases, some buckets in the hash file are split and the records in these buckets may be moved to the new buckets. The density of the hash file becomes lower and the average I/Os per probe should also decrease or at least remain the same. Given a desired number of I/Os, there is only one unknown variable in Equation 3.10 which is N, but it is not easy to solve the equation directly. Algorithm 2 presents the steps for finding an optimal value of N. Parameter *Max* indicates the maximum value that must be examined to find an optimal solution to the equation. In practical cases, our desired number of I/Os cannot be too small or too large. Thus, we can do a binary search for $N \in [1, n]$; the extreme case $N = 1$ is when all records are inserted into a single bucket, and the extreme case $N = n$ is when each bucket has only 1 record on average.

Algorithm 2 Estimate the number of hash buckets for random data

n: {number of records}
I/O: {a user-supplied value which is greater than 1}
$f(N)$:{the average I/O cost to retrieve one record}

Let $f(N) = (T_1(N) + T_2(N))/n$

Do a binary search for $f(N)$ in the interval of [1,Max] to find the smallest N where $f(N) <= I/O$.

Lemma 4. *Algorithm 2 correctly finds an optimal number of buckets N.*

The proof can be easily derived from Lemma 3. As a proof of concept, we randomly generated 10 million records, each with a 32-bit random hash value, and applied Algorithm 2 to find a hash layout with a user-specified bound on the expected number of I/Os for a hash probe. We also actually built the hash file and counted the average number of buckets that are accessed for a hash probe. The results are shown in Table 3.1. Given a user-supplied I/O, our predicted layout has a real average I/O that is very close to the user-supplied I/O.

Table 3.1: Quality of our layout estimation for random data

User-supplied I/O	1.05	1.10	1.20
Real I/O	1.052	1.100	1.223

3.1.2 Layout Estimation for Skewed Data

The distribution of records in the address space can vary with the hash function that is chosen and the data set that is being loaded. In general, we may not be able to guarantee that a given hash function converts the keys into hash values randomly. Therefore, using our earlier analysis to estimate a hash layout is questionable. A solution is to construct a histogram for each candidate class of equivalent layouts, with cell i of the histogram showing the tally of records that are going to be mapped to bucket i. Given that each bucket has a fixed size, it is easy to find the exact number of both the overflow records and the empty slots.

A problem here is that we may not have enough memory to construct one such histogram. Given a layout that uses r bits for addressing, the histogram must have 2^r cells to accurately show the tallies of records in each bucket. For $r = 40$, for instance, if we use two bytes to record a tally, we will need 2 TB

of memory to construct a histogram. Clearly, this is not feasible for a large data set. An alternative is to choose $r_i \leq r$ initial bits for addressing and construct a histogram with 2^{r_i} cells. The value of r_i can be chosen such that the histogram can fit in the available memory. Each cell of the histogram is associated with 2^{r-r_i} buckets and keeps the total count of the records that are mapped to those buckets. If we can assume that the records that are mapped to a histogram cell are randomly distributed within the buckets associated to the cell, then we can use our earlier analysis for records within a cell. In particular, Equation 3.10 can be used to estimate the number of I/Os needed to retrieve the records within a cell. The histogram has 2^{r_i} cells and N denotes the number of buckets that are allocated for the entire hash file, hence the number of buckets for each cell is $N/2^{r_i}$. Suppose the average I/O cost to retrieve a record from cell j is $f_j()$; the average number of I/Os for the entire hash file is

$$f(N) = \sum_{j=1}^{2^{r_i}} \frac{n_j}{n} * f_j(N/2^{r_i})$$

where n_j is the number of records in the j^{th} cell and n is the number of records in the whole histogram.

Algorithm 3 presents the steps for finding an optimal value of N for skewed data. Since we know the tally for each cell, $f_j()$ can be computed using Equation 3.10 for each histogram cell j. Parameter *Max* indicates the maximum value that must be examined to find an optimal solution to the equation. As before, *Max* can be set to n. Clearly, the larger the size of the histogram, the more accurate our estimation of the skewness of the data. The limit is that the size of the histogram cannot exceed the size of the allowable memory.

As a proof of concept, we generated two data sets with 10 million records each. Each record of the data set had a 32-bit hash value. The first 27 bits of the records in these two data sets are completely random and the remaining 5

Algorithm 3 Estimate the number of hash buckets for skewed data

I/O: {a user-supplied value which is greater than 1}
r_i: {initial number of bits for addressing}
r: {number of bits for addressing}
n: {total number of records}
n_j: {number of records in the j^{th} cell in the histogram}
f_j: {the average I/O cost of retrieving a record from the j^{th} cell of the histogram}

Construct a histogram of the data set using r_i initial bits for addressing.

Let $f(N) = \sum_{j=1}^{2^r} \frac{n_j}{n} * f_j(N/2^{r_i})$

Do a binary search for $f(N)$ in the interval of [1,Max] to find the smallest N where $f(N) <= I/O$.

---

bits are got using two different methods. For one data set, 5 least significant bits of the hash values generated using a Zipfian distribution [13] in which the frequency of the k^{th} value was proportional to $(1/k)^{\theta}$, where $0 < \theta < 1$ was the skew.

For the other data set, 5 least significant bits of the hash values were generated using a binomial distribution with a biased coin, which means heads and tails have different probabilities when we flip a coin.

Algorithm 3 was applied on these data sets varying the initial number of bits r_i. The desired number of I/Os or the user-input I/Os was set to 1.10, but it could have been equally set to any other number. Figures 3.2 and 3.3 show that building a histogram for skewed data helps, and the benefit increases with the skewness of the data. Furthermore, the more space is allocated for a histogram, the closer our estimation is to the real number of I/Os. Even a small histogram can also reduce the effect of the skewness on the final result. In the example zipf distribution, a $2^3 = 8$ bytes histogram using 3 initial bits

for addressing helps a lot which can reduce the gap between the user's input I/O and real I/O from 0.52 to 0.27 when the skew $\theta = 0.8$.

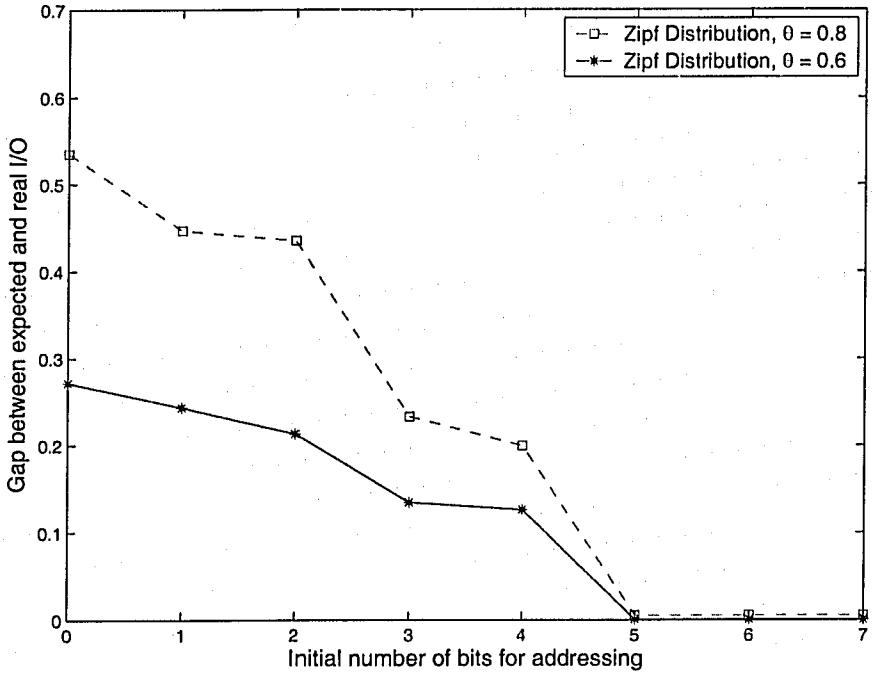

Figure 3.2: Layout Estimation for Zipf Distributed Data

3.2 Layout Parameter Estimation based on access cost and storage overhead

Instead of using a user-specified I/O value to predict the hash layout, another method is to use different penalty weights to balance the access cost and the wasted storage space and then predict a hash layout. This method has the benefit when the users do not know what I/O value they need. The overflow buckets and empty slots are assigned different penalty scores. We use a user-tunable penalty function and seek a layout that optimizes this function. Since the penalty weights of empty buckets and overflow slots are user-defined values, the hash layouts can easily shrink and grow with different parameters: increasing the penalty weights of empty buckets may cause a compact hash

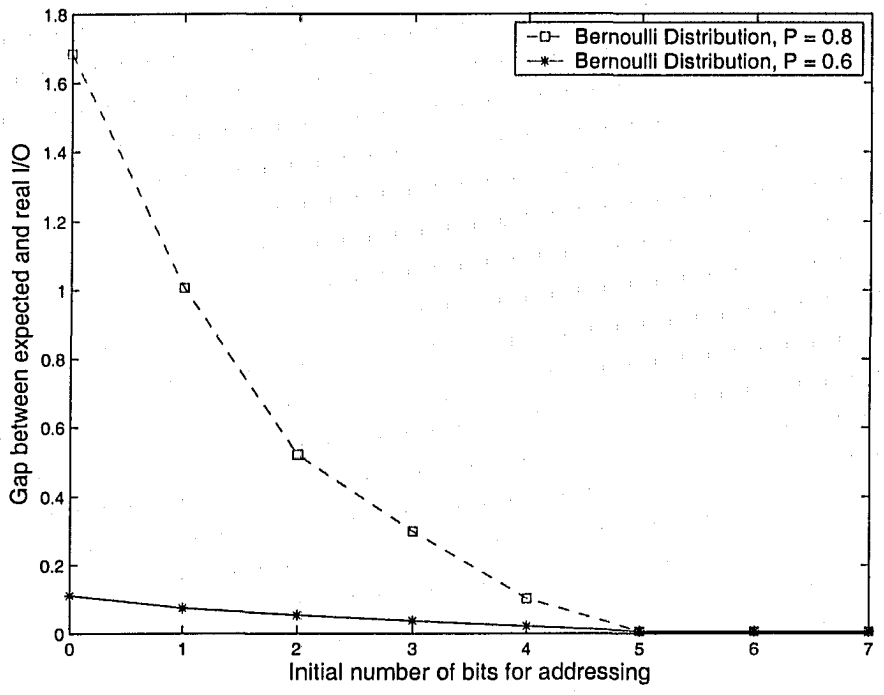

Figure 3.3: Layout Estimation for Bernoulli Distributed Data

file and vice versa. Also, our predicted layout is the hash file at the beginning (or the end) of a generation, hence the number of primary buckets (excluding overflow buckets) is 2^r where r is the number of bits used for addressing.

Suppose an empty slot in a bucket is penalized by E/L and an overflow record is penalized by O/L where E and O are some user-defined scores; we discuss some settings of E and O at the end of this section. Our goal is to find a layout that minimizes the overall penalty score. We do our estimation in the following two cases: (1) the hash function distributes the records uniformly at random in the address space, and (2) the hash function is skewed toward a few addresses.

3.2.1 Layout Estimation for Randomly Distributed Data

Let N denote the number of primary buckets in the hash file. We want to find a value of N that optimizes the overall penalty score. Suppose the hash

function is chosen in a way that it distributes the records randomly in the address space.

An approximation of this function which is easier to compute for large values of n and N can be given using the following Poisson distribution:

$$P(x) = e^{-\lambda}\frac{\lambda^x}{x!} \tag{3.11}$$

where $\lambda = n/N$. The parameter λ can be treated as the fill factor of the hash file, which is the ratio of the number of records and the number of buckets.

Using Equation 3.11, we can estimate both the number of empty slots and the number of overflow records in the final layout. The expected number of buckets with i records in them is $N * P(i)$, and the number of empty slots in one such bucket is $L - i$. Hence, the penalty score for empty slots can be computed as

$$S_e(N) = \frac{E}{L}\sum_{i=0}^{L}(L-i) * N * P(i). \tag{3.12}$$

Similarly the expected number of buckets with i overflow records mapped to them is $N * P(L+i)$. Therefore, the penalty score for overflow records can be computed as

$$S_o(N) = \frac{O}{L}\sum_{i=1}^{\infty} i * N * P(L+i). \tag{3.13}$$

The overall penalty score for a given value of N is

$$S(N) = S_e(N) + S_o(N). \tag{3.14}$$

If the number of records to be loaded is fixed, increasing the number of hash buckets N should cause more empty buckets and less overflow buckets. Therefore, S_e is a monotonic decreasing function and S_o is a monotonic increasing function of the fill factor. Fig. 3.4 and Fig. 3.5 listed below are S_e and S_o functions for 1 million random records.

Fig. 3.6 is the overall penalty score function for 1 million random records. We want to find a value of N that minimizes the overall penalty score. Consider

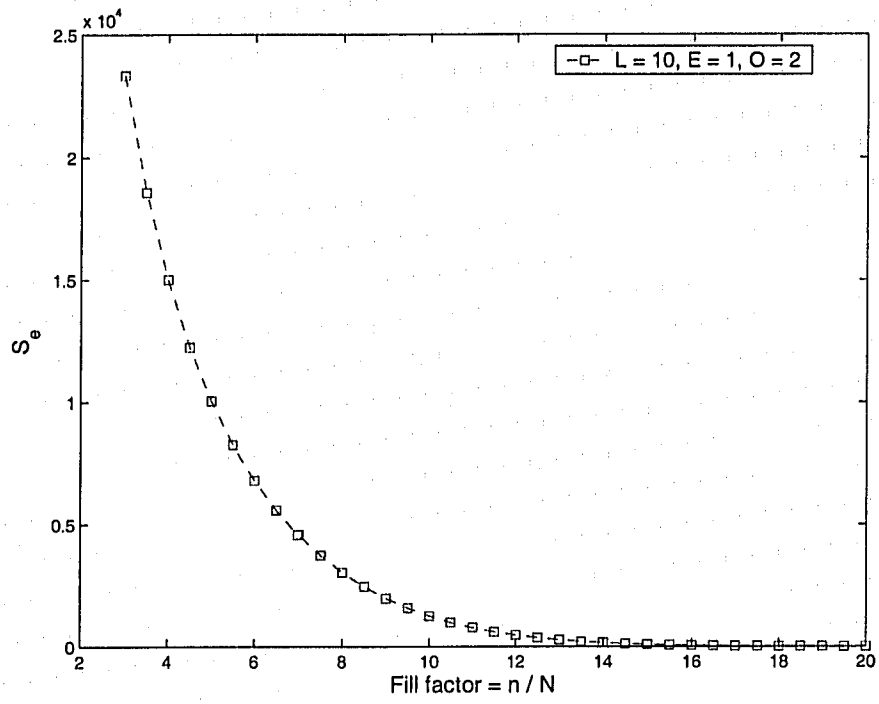

Figure 3.4: Penalty score for empty slots of 1 million random records

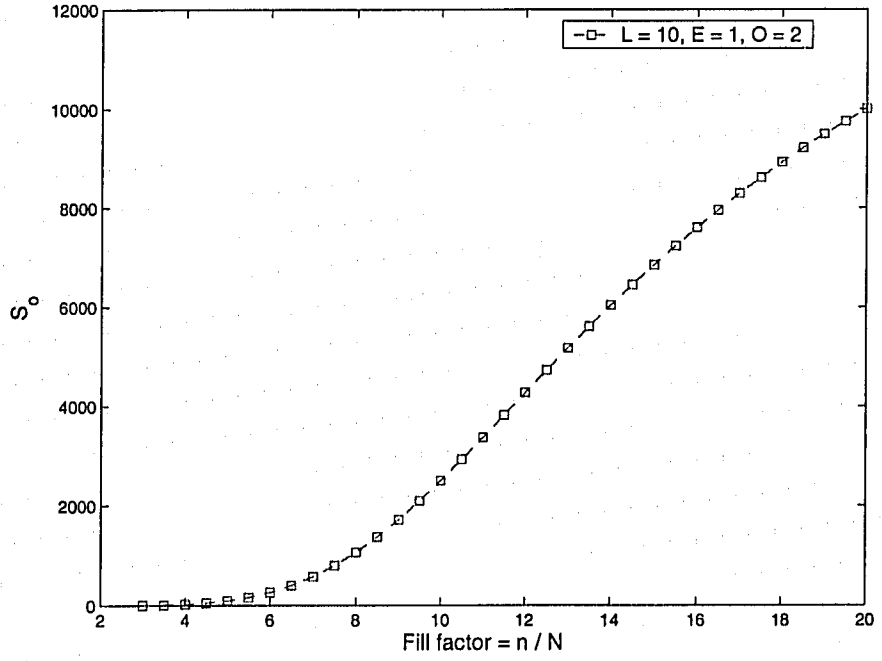

Figure 3.5: Penalty score for overflow buckets of 1 million random records

the case where $N = 2^r$ and r is the number of bits used for addressing. Since in practical settings, the number of bits used for addressing is usually small (e.g. less than 50), it is not hard to compute $S(2^r)$ for all such values of r using Eq. 3.14 and find the value of r that gives the minimum penalty score. We refer to this value of r as the optimal number of bits for addressing.

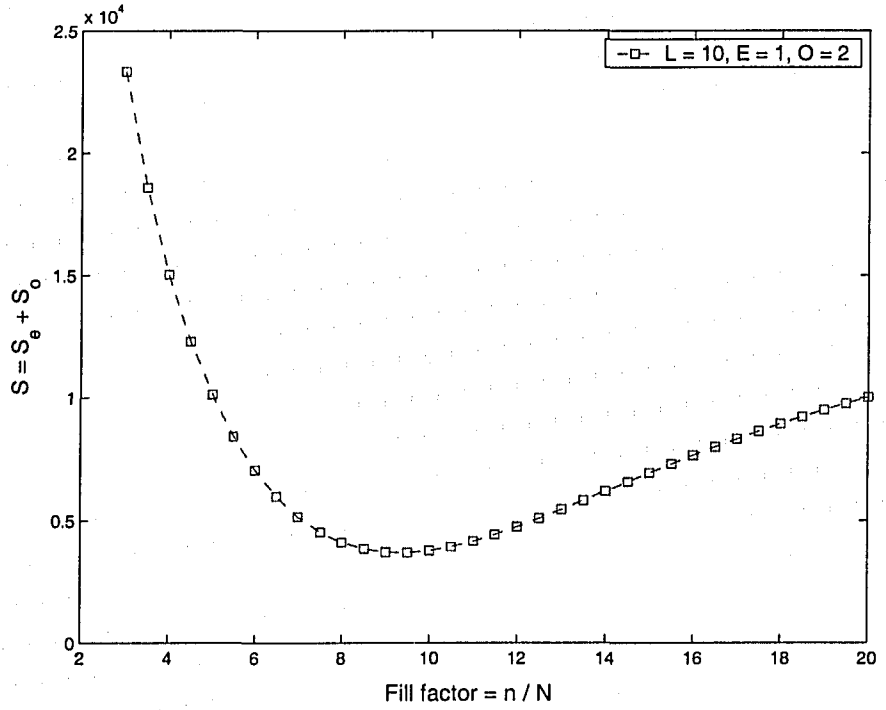

Figure 3.6: Overall penalty score of 1 million random records

Algorithm 4 presents the steps for finding an optimal value of N. Parameter *Max* in Algorithm 4 indicates the maximum number of bits that must be examined to find an optimal solution to the equation. This number can vary depending on the values of E and O. In an extreme case where there is no penalty for overflow records, i.e. $O = 0$ and $E > 0$, the optimal value of r is zero; hence *Max* can be set to zero. In another extreme case where there is no penalty for empty slots, i.e. $E = 0$ and $O > 0$, any value of r that results in no overflow records is optimal; hence *Max* must be large enough to guarantee

Algorithm 4 Optimizing the penalty score for randomly distributed data

```
min_score ← ∞ {minimum penalty score}
r ← 0  {number of bits for addressing}

for i = 0 to Max do
  S = S_e(2^i) + S_o(2^i)
  if S < min_score then
    min_score = S
    r = i
  end if
end for
N = 2^r
```

that there is no overflow records. In practical settings where both E and O are positive values, Max cannot be a large number.

3.2.2 Layout Estimation for Skewed Data

If the given hash function can not distribute the records randomly, a histogram is also used for approximating the number of primary hash buckets in the final hash layout. We construct a histogram for each candidate layout, with cell i of the histogram showing the tally of records that are going to be mapped to bucket i. Given that each bucket has a fixed size, it is easy to find the exact number of both the overflow records and the empty slots. We can calculate a penalty score using Eq. 3.14 and pick the layout with the minimum score.

Similar to our previous method, the histogram itself may be too large to fit in memory. An alternative is to choose $r_i \leq r$ initial bits for addressing and construct a histogram with 2^{r_i} cells. The value of r_i can be chosen such that the histogram can fit in main memory. Each cell of the histogram is associated with 2^{r-r_i} buckets and keeps the total count of the records that are mapped to those buckets. If we can assume that the records mapped to each cell of the histogram are randomly distributed within the buckets associated to the cell, then we can use the Poisson distribution to predict the number of both overflow

records and empty slots within each cell. The sum of the penalty scores of all the cells gives the penalty score of the layout. Algorithm 5 presents our estimation method in pseudo code.

Algorithm 5 Optimizing the penalty score for skewed data

$min\_score \leftarrow \infty$ {minimum penalty score}
r_i : {initial number of bits for addressing}
r_m : {number of more bits for addressing}
$r = r_i + r_m$ {number of bits for addressing}

Construct a histogram of the data set using r_i initial bits for addressing

for $j = 0$ to $Max - r_i$ **do**
  $S \leftarrow 0$
  for each cell c of the histogram **do**
    Set n to the number of records mapped to cell c;
    $S+ = S_e(2^j) + S_o(2^j)$
  end for
  if $S < min\_score$ **then**
    $min\_score = S$
    $r_m = j$
  end if
end for
$r = r_i + r_m$
$N = 2^r$

Clearly, the larger the size of the histogram, the more accurate our estimation of the skewness of the data. The limit is that the size of the histogram cannot exceed the size of the memory. Also the step to find the minimum penalty score requires scanning the histogram many times and can be costly. An alternative is to examine only a limited number of all possible values for for N.

An optimal layout in both algorithms 4 and 5 can vary depending on the values of E and O. If the storage space is more precious, we probably prefer more overflows records than empty positions; thus parameter O can be reduced and E can be increased. This can save some storage space. However, if the

access time is more important, we probably prefer more empty positions than overflow buckets; hence O can be increased and E can be reduced.

3.2.3 Parameter settings of bulk loading

Our bulk loading algorithm uses the parameters E and O to balance the access cost and the storage overhead. In an experiment to show the effects of different settings of these variables, we tried to estimate the best layout using our estimation techniques in Section 3 for different settings of these variables.

We used a data set with 10 million records and applied Algorithm 5 to estimate the number of buckets. The capacity of a bucket was set to 10 records (i.e. $L = 10$) and E and O varied. We first fixed E to 1 and varied O from 1 to 10. As is shown in Figure 3.7, when the penalty score for an overflow record is increasing, the number of buckets N and as a result the average number of bits used for addressing $r = log_2(N)$ is gradually increasing.

Figure 3.8 shows the scenario where O is fixed to 1 and E is varied to indicate that an empty slot is penalized more than an overflow record. Both the number of buckets and the average number of bits used for addressing are either decreasing or remain the same, hence the hash file becomes more compact.

To measure the effect of these parameter settings on the actual query performance, we loaded the data using our bulk loading algorithm and with the number of bits r set to 19, 20 and 21. This resulted in three hash files. We selected 100,000 random queries from the data set and posed them to each one of the hash files. Each query was posed 100 times and the average response time was recorded. For each hash file, we measured the average running time of a query and the average number of I/Os. To measure the number of I/Os, we counted the number of buckets that we needed to access for each query. Since there could not be much buffering effects for randomly-selected queries,

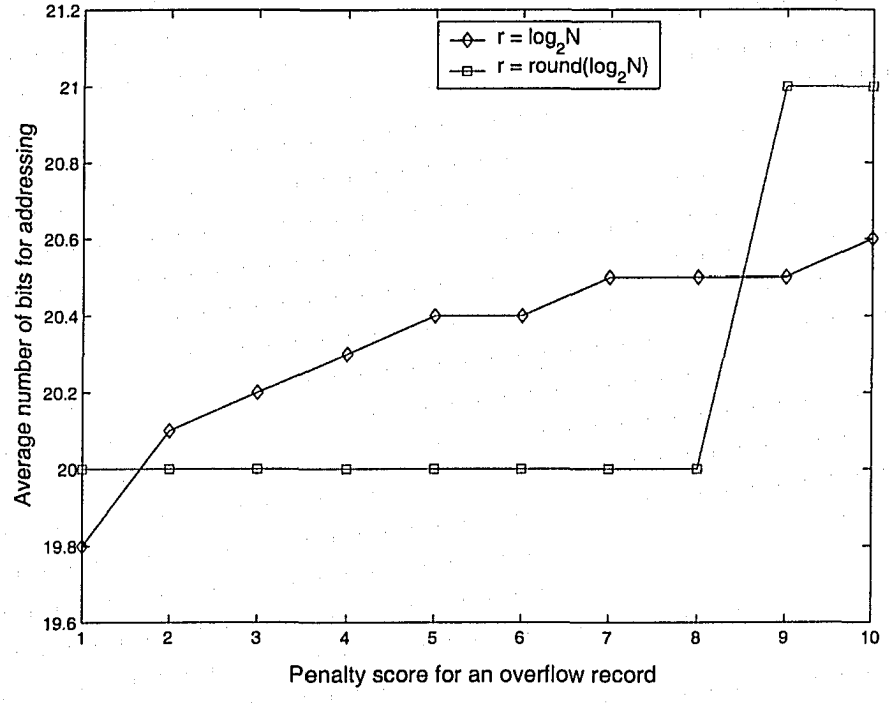

Figure 3.7: Average number of bits used for addressing varying the penalty for an overflow record

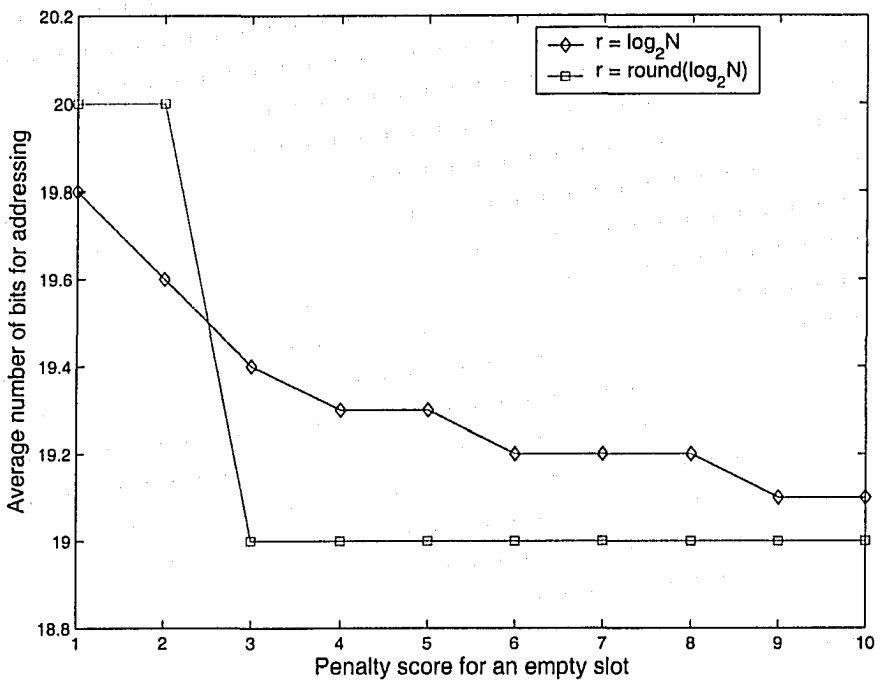

Figure 3.8: Average number of bits used for addressing varying the penalty for an empty slot

the number of buckets that needed to be accessed was a good indication of the number of I/Os. As is shown in Table 3.2, when the number of bits for addressing is increased from 19 to 21, the hash file becomes more query efficient; both the running time and the number of I/Os are dropped, but this is for the cost of some additional disk space.

Table 3.2: Query cost varying the number of bits for addressing

r	19	20	21
running time (msec)	0.082	0.072	0.070
# of I/Os	1.54	1.10	1.00
file size	1.21 GB	1.37 GB	2.48 GB

These experiments confirm that setting the parameters E and O is indeed important in balancing the access cost and the wasted storage space, and that the performance of the hash file can be tuned as it is desired by properly setting these variables.

In the rest of the experiments, we set $E = 2$ and $O = 1$ to indicate that an empty slot must be penalized twice as much as an overflow record; this setting is expected to produce a rather compact hash file. We also set the number of bits for addressing $r = round(log_2(N))$ where N is the estimated number of buckets. The capacity of a bucket $L = 10$ and the number of records n can vary from one experiment to next.

Chapter 4

Improving Loading in Berkeley DB

There are efficient implementations of linear hashing in practice (e.g. Berkeley DB [5]). and dynahash [1]. Berkeley DB supports linear hashing through its so-called extended linear hashing [17]; it provides functions to construct and search a linear hash index but does not support bulk loading. To use our bulk loading algorithm, we need to construct a linear hash file that can be correctly recognized by the library. Without knowing the detailed file format and the overflow handling method that is used, it is not easy to construct one such file. In this section, we take a different approach to bulk loading; we treat the library as a black box and try to efficiently load data using the external function calls that are provided. The function calls that are needed include a function to insert a record and a function that takes a key and returns its hash value. This method of loading in general can be useful in some of the systems that support linear hashing but do not expose the details of their implementations.

Berkeley DB provides a load utility, called *db_load*, which when used to load data, reads one record at a time and adds this record into the hash table. Below we list a few parameters of *db_load* utility that are useful in our experiments.

-f Read from the specified input file instead of from the standard input.

-T The -T option allows non-Berkeley DB applications to easily load text files into databases.

-t Specify the underlying access method. If no -t option is specified, the database will be loaded into a database of the same type as was dumped; for example, a Hash database will be created if a Hash database was dumped. In our experiment, obviously we set it to "hash".

The following keywords are supported for the -c command-line option to the db_load utility.

db_pagesize (number) The size of database pages, in bytes. The minimum page size is 512 bytes, the maximum page size is 64K bytes, and the page size must be a power of two. In our experiment, we use the underlying linux filesystem I/O block size, which is $4096bytes = 4KB$.

h_ffactor (number) The fill factor of the Hash database. The fill factor is an approximation of the number of records allowed to store in one bucket. In our experiment, since we know the size of the keys and data in our data set, we set the fill factor using Berkeley DB's recommended formula:

$$(pagesize - 32)/(average\_key\_size + average\_data\_size + 8)$$

h_nelem (number) The size of the Hash database. In our experiment, this parameter is set to the number of records in the data set.

To integrate our sorting procedure into this utility, we buffer the input and sort the records inside the buffer based on their reversed hash values. The buffering partitions the input into smaller chunks, and the sorting reorders the records in each partition so that the records in the same partition which belong to the same or adjacent buckets are grouped together. Algorithm 6 presents our modified version of the load utility. We use linux's qsort command to sort

the records in the buffer.

Algorithm 6 Our modified version of *db_load* in Berkeley DB

```
Initialize the memory buffer
while there are more records do
  Read a record R from the data set 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 bulk loading performance. The larger the buffer, the more records will be grouped according to their positions in the hash table. We may want to allocate as much buffer space as possible but we are often limited by the size of the main memory. Furthermore, sorting a few small partitions can be faster than sorting the whole data set. Our experiments (reported in the next section) show that even adding a small buffer can significantly improve the performance. The buffer used for sorting is always more effective than the I/O cache of the same size.

An alternative is to sort all the records, based on their reversed hash values, using an external sort utility and pipe the result to the original *db_load* utility. This has the benefit that the input ordering pretty much corresponds to the ordering of the records in the hash file. We compare this solution to the partial sorting buffering method in our experiments.

Chapter 5

Experiments

To show the scalability and the performance improvements of our algorithms, we conducted experiments comparing our bulk loading to both our implementation of a naive load algorithm and the loading in Berkeley DB.

Our experiments were conducted on a real data set of URLs. The data was extracted from a set of crawled pages in the Internet Archive ([2], [4]) collection. Attached to each URL was a 64-bit unique fingerprint which was produced using Rabin's fingerprinting scheme [15]. To experiment with larger keys, 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 character string for data values. We also tried using URLs as our keys but the result was pretty much the same and they are not reported here. The records in our data set look as follows:

Keys Data

.....

00000000312E637A abcdefghijklmn.....

00000000382E746F abcdefghijklmn.....

00000000672E746F abcdefghijklmn.....

000000006C2E6E75 abcdefghijklmn.....

00000000762E6E75 abcdefghijklmn.....

00000032712E746F abcdefghijklmn.....

00000032752E746F abcdefghijklmn.....

00000038332E6E75 abcdefghijklmn.....

0000004FDEC99112 abcdefghijklmn.....

00000061312E706C abcdefghijklmn.....

.....

All our experiments were conducted on a Pentium 4 machine running Red Hat 9, with a speed of 3.0GHz, a memory of 2 GB, and a stripped array of three 7200 RPM IDE disks. We used the version 4.2.52 of Berkeley DB, which was the latest at the time of running our experiments.

For sorting in our algorithms, we had the option to sort the data by the right number of bits after the right number of bits was estimated. However, we decided not to do it for a few reasons. First, this required a tight integration of our layout estimation with our sorting if we wanted to avoid an extra scan of the data. Second, there was not much improvement in term of performance when the number of bits used for sorting was less than the full hash values. For instance, our experiments with external sorting 180 million 130-byte records showed that a sort based on 16 bits takes 85 minutes whereas a sort based on 64 bits takes 87 minutes. Third, for the partial-sorting algorithm (Alg. 6), using our approximation algorithm, the number of bits for addressing of the records in the buffer and that of records in the final hash layout are different. What's more, even for the records in the buffer, the number of bits for addressing is directly affected by the number of records in the buffer. Then, using different sizes of sorting buffers also requires different numbers of bits for sorting. Sorting the records based on the reversed hash value instead of the right number of bits can avoid a lot of troubles and the trivial performance difference can be omitted. Therefore, our reported experiments here all use

sorting based on the full hash values.

For both efficiency and scalability reasons, we used external sorting when the entire data set needed to be sorted. In our experiments, we use linux's sort command.

5.1 Performance comparison to naive loading

As a baseline comparison to our bulk loading, we implemented a naive loading of a linear hash file which inserts one record at a time, and compared its performance to our implementation of the loading algorithm (Alg. 1, using a user-specified I/O of 1.10). Both implementations use the same file format and overflow handling method.

We varied the size of the data set from 1 million to 50 million records. We couldn't run the naive loading for larger data sets; it was taking already more than 55 hours to run it with 50 million records about 5.9 GB. The result of the comparison is reported in Fig. 5.1 and Table 5.1.

The X axis in the figure is the number of records in millions, and the Y axis is the improvement factor, in terms of running time, of our bulk loading algorithm compared to the naive loading. Loading 10 million records, for instance, using our bulk loading algorithm takes 3 minutes and 16 seconds whereas it takes 129 minutes and 55 seconds to load the same data set using the naive algorithm. For 50 million records, using our bulk loading algorithm takes 27 minutes and 4 seconds whereas the naive algorithm needs 3333 minutes and 18 seconds. Our algorithm is 123 times faster. Generally speaking, our bulk loading algorithm outperforms the naive loading by two orders of magnitude, and its performance even gets better for larger data sets. The reason is there are huge numbers of bucket splitting and record movements (read and write) in the naive loading. However, our bulk loading algorithm performs external sorting and sequential writing. All the splitting and movements are avoided.

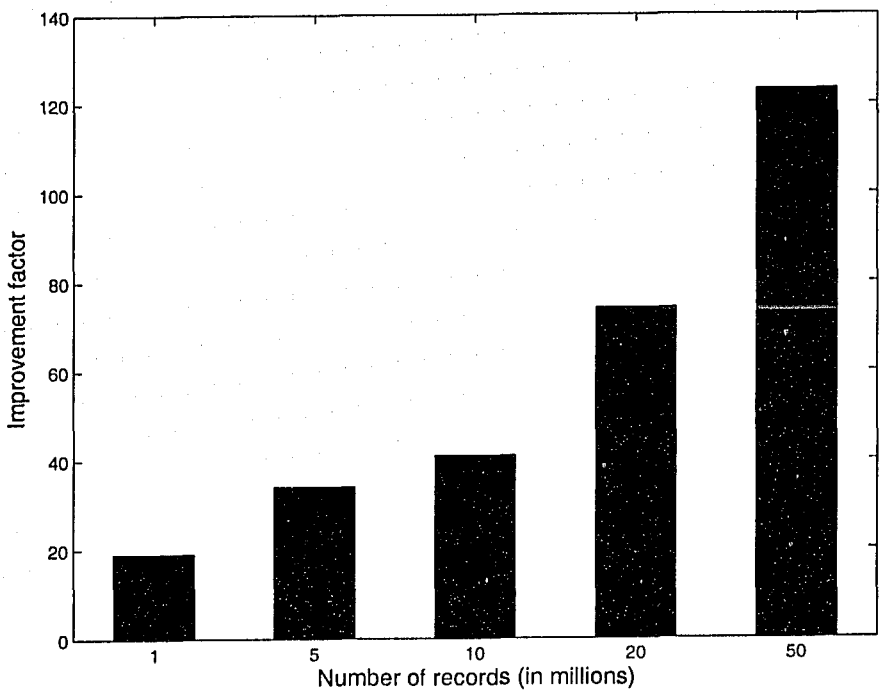

Figure 5.1: Improvement factor varying the number of records for our bulk loading compared to the naive loading

Table 5.1: Running time of our bulk loading compared to a naive loading

Number of records	1M	5M	10M	20M	50M
Size of data set	118M	590M	1.18G	2.36G	5.9G
Bulk loading	0.17 min	1.40 min	3.27 min	10.15 min	27.07 min
Naive loading	3.17 min	49.80 min	129.92 min	752.43 min	3333.30 min

5.2 Performance comparison to the *db_load* utility in Berkeley DB

As another baseline for our comparison, we used the native Berkeley DB load utility and compared its performance to that of our modified version of the same utility and also our bulk loading algorithm.

5.2.1 Scalability with the size of the data set

We varied the size of the data set from 1 million to 20 million records and measured the loading time. The size of the sorting buffer in our modified version of the *db_load* utility was set to 300MB (our next experiment shows how the buffer size can affect the load performance). On the other hand, Berkeley DB automatically allocated 1 MB I/O cache to *db_load* utility. The total buffer size of our modified *db_load* utility was 301MB. To make the comparisons fair, we also set the I/O cache of the native *db_load* utility to 301 MB. In the following experiment, all the parameters including db_pagesize, h_ffactor and h_nelem of these two utilities are set to their default values by Berkeley DB. We couldn't run the experiments for larger data sets (such as 50 million records) with the original *db_load* utility of Berkeley DB because of the low performance of Berkeley DB in this case.

The result of the experiment is shown in Figure 5.2. When the data set is only 1 million records (or 118 MB), the I/O cache allocated to the native version of the *db_load* utility is large enough to keep the whole data set and therefore, the whole hash table is built in the memory. In this scenario, our modified version of the *db_load* utility doesn't outperform the native *db_load* utility of Berkeley DB. When the data set is 5 million, the size of the data set is more than 590 MB, and the whole data cannot fit in the sorting buffer of our modified *db_load* utility or the I/O cache of the native *db_load* utility of

Table 5.2: Loading records in the hash tables with different number of records

Number of Records	1M	5M	10M	20M
Running Time of our approach	1.53 min	13.20 min	59.02 min	299.08 min
Running Time of Berkeley DB	7.75 min	86.27 min	216.88 min	893.75 min

Berkeley DB. The results shows that sorting the records in the buffer based on the reversed hash values can improve the performance by a factor of 1.5. When the data set contains more than 10 million records, our experiment shows that our modified *db_load* outperforms the native *db_load* utility in Berkeley DB 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 data set with 20 million records while native *db_load* utility in Berkeley DB requires 1682 minutes and 1 seconds. This result shows the sorting-buffer is more effective than the I/O cache when loading using Berkeley DB. When the size of the memory is limited, we should allocate as much sorting-buffer as possible.

In Berkeley DB, the number of records to be loaded by the *db_load* utility can be specified by a parameter called "h_nelem." When "h_nelem" is set, the *db_load* utility attempts to build the whole empty hash table at the beginning instead of using the dynamic hashing strategy. In our experiments, however, we did not notice any improvements on the performance of the native *db_load* utility after setting "h_nelem" in advance to load a data set with unsorted records.

5.2.2 The effect of buffer size on the performance

As shown in Figure 5.2, when the data set cannot be fully loaded into the memory, the sort buffer is always more effective than the I/O cache of the same size within the native *db_load* utility. In another experiment to measure the effect of the sort buffer on the performance of our modified *db_load* utility,

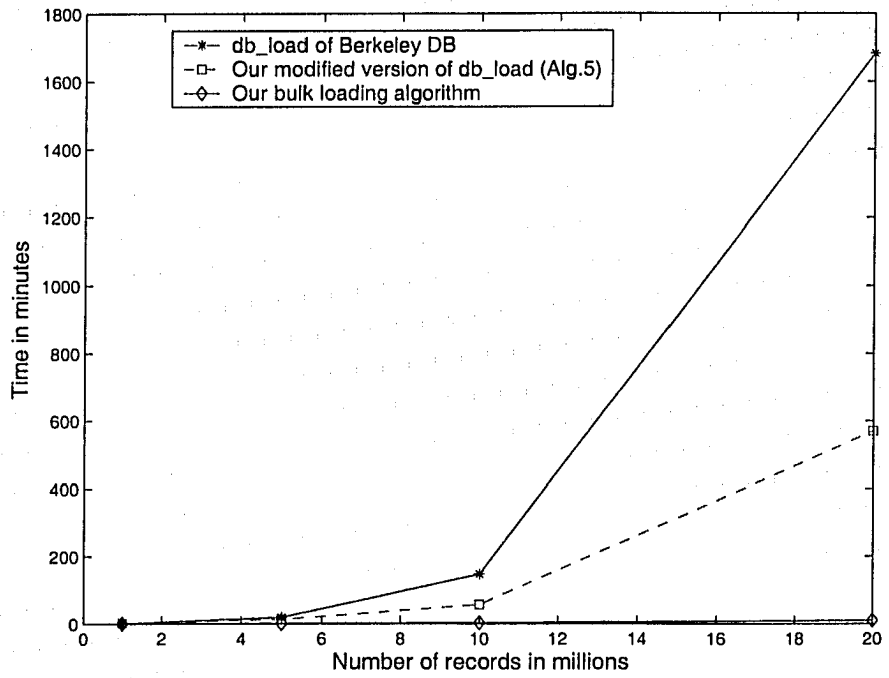

Figure 5.2: Running time varying the number of records

Table 5.3: Loading the hash index with different buffer sizes

Buffer Size	10M	50M	100M	300M	500M	1G
Total Cost (min)	106.62	39.08	28.68	23.65	18.15	19.53

we fixed the size of the data set to 10 million records and varied the sort buffer size from 100 MB to 1 GB. Each record contained a 16-bytes key and a 50-bytes data field. The default I/O cache size of *db_load* was 1 MB. As is shown in Figure 5.3, increasing the buffer size improves the performance of the new *db_load* utility up to a point where the whole data set can be fit in buffer. After this point, the performance remains the same.

It is clear that the sort buffer size has a significant impact on the running time of the algorithm. When the buffer size is increased from 10MB to 500MB, the loading time is decreased considerably. This is because when the buffer size is too small, data is sorted only within small segments and we can't guarantee all the records which finally belong to the same bucket are grouped together.

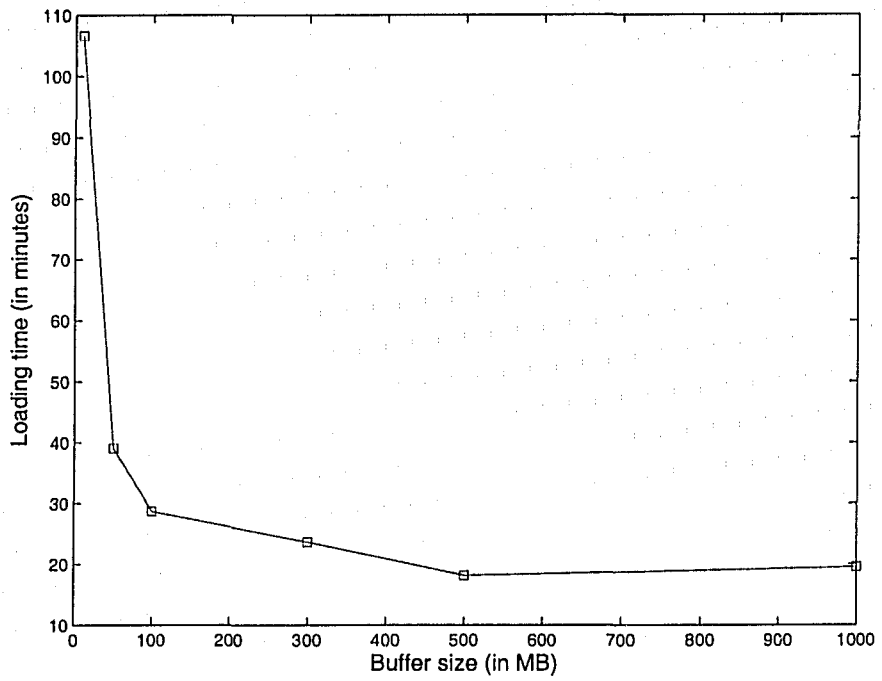

Figure 5.3: Running time varying the buffer size for our modified *db_load*

However, as is shown in the figure, even allocating a modest-sized buffer (e.g. 50MB) can significantly improve the performance. In the extreme case where the buffer size is greater than 600MB, the buffer is large enough to load all the records. The hash table can be built in memory, and increasing the buffer size has no more impact.

5.2.3 Sorting data in advance

In another experiment, we sorted the records based on their reversed hash values using the external sort command in Linux and loaded the sorted data using the native *db_load* utility with and without "h_nelem" setting. The results are compared to that of our bulk loading algorithm.

As is shown in Figure 5.4, setting the number of records "h_nelem" this time can improve the performance of the native *db_load* utility by at least a factor of 3. When the data set contained 20 million records, the native *db_load*

utility with a pre-specified number of elements took an order of magnitude less time than the case when "h_nelem" was not provided. Combining the results with those in Figure 5.2, we can conclude that setting the number of records in advance can increase the performance only when the records are sorted by their reversed hash values. This is because it is after this ordering that the load order becomes the same as the order in which the records sit in the hash table. Therefore, records movements between hash buckets are avoided.

In this experiment, we've also tried to set the "h_ffactor" which is called the fill factor that represents the density of the hash file. Berkeley DB suggest to use the following formula to calculate the average number of records in one page. However, we did not notice any obvious improvement after setting the fill factor.

$$(pagesize - 32)/(average\_key\_size + average\_data\_size + 8)$$

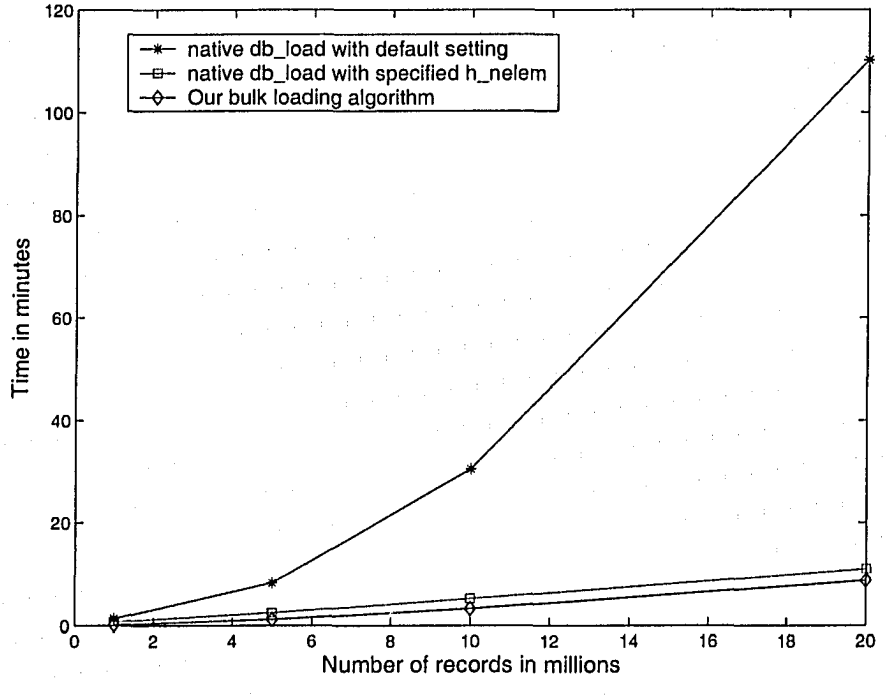

Figure 5.4: Loading with sorted records

Table 5.4: Loading with sorted records

Number of recrods	1M	5M	10M	20M
Total Cost with specified h_nelem	0.74 min	2.47 min	5.2 min	10.96 min
Total Cost with default setting	1.35 min	8.29 min	30.51 min	110.17 min

The performance of the native *db_load* utility with a pre-specified number of records is also comparable with that of our bulk loading algorithm. After the records are sorted based on their reversed hash values, both methods just read the records from the data set one by one and then write them into the hash table. Buckets split and records movements don't occur. The only difference is that our bulk loading algorithm use Alg. 3 to estimate the layout before really building it.

Based on the above experiments with Berkeley DB, to load a data set into a linear hash index with Berkeley DB should follow the following 2 rules:

- If we want to load the records incrementally when they are received, the sorting-buffer strategy should be use. The size of the allocated sorting-buffer should be as large as possible.
- If we want to load the entire data set once, we need to sort all the records based on their reversed hash values. Then load the sorted data set using *db_load* utility with the number of records specified.

5.3 Quality of the hash file

It is clear from our earlier experiments that our bulk loading algorithm can significantly reduce the loading time, but a question that arise is how a hash file generated using our bulk loading algorithm compares to a hash file constructed, for instance, using Berkeley DB. Reducing the loading time may not be that important if the query response time is much longer.

We have proposed two cost functions to approximate the number of hash buckets in the final hash layout or the number of bits for addressing when hashing. In our first method, we use a user-tunable penalty function and try to find a hash layout with a user-specified average I/O for retrieving a record. In our second method, we use E and O as the penalty score for an empty slot and an overflow record. Then we define a user-tunable penalty function to find a good trade-off between the access cost and storage overhead.

We did some experiments using both of the above methods. In the experiment using the input I/O method to compare the query response time, we loaded 10 million records into a hash file using both our bulk loading algorithm and our naive loading (as discussed earlier). To make a fair query cost comparison, we forced the hash files in both cases to have the same load factors. We first generated a hash file using our bulk loading algorithm, with average $I/O = 1.10$ per probe and calculated the load factor. We then used the same load factor to generate another hash file for the same data set but using the naive loading algorithm.

We randomly selected 100,000 queries from the data set and posed them to the two hash files. For each query, we measured the running time and calculated the average running time for the 100,000 queries. Table 5.5 shows the result of the comparison.

As is shown in Table 5.5, when the size of the data set and the load factor are fixed, the size of the two hash files are also the same. More importantly, for each query the average number of I/O accesses over our hash file is the same as that over the hash file built using the naive loading. The average query response times for the two hash files are also nearly the same. From this experiment, we can say that the quality of the hash file built using our bulk loading algorithm is very close (if not identical) to the hash file built using the naive loading.

Table 5.5: Query cost comparison between the hash files (user-specified I/O)

	Size of the hash file	Load factor	Average query cost	Average I/O accesses
Bulk loading	1.37 GB	0.95	0.072 msec	1.10
Naive loading	1.37 GB	0.95	0.071 msec	1.10

We also compared the quality of the hash file built using bulk loading to the one generated using the *db_load* utility of Berkeley DB. The load factors for the two hash files were fixed to 0.47. This corresponded to $I/O = 1.04$ in our bulk loading algorithm. The results are listed in Table 5.6. We can see that the query average response time for our algorithm is much more less than that of Berkeley DB. The reason is our algorithm and Berkeley DB use different file formats. What's more, Berkeley DB seems to do some additional caching works while querying. When we ran the queries for the first time in Berkeley DB, it took more than 8 minutes to process 100,000 queries. However, it took only 1 second if we ran all the 100,000 queries again.

Table 5.6: Query cost comparison with Berkeley DB (user-specified I/O)

	Size of the hash file	Load factor	Average query cost
Bulk loading	2.48 GB	0.47	0.070 msec
Berkeley DB	2.62 GB	0.47	4.80 msec

Our two methods to predict the hash layout before loading only can affect the number of bits for addressing. Their bulk loading procedures are exactly the same. Therefore, if the load factors are the same, the hash file using penalty score method of our bulking algorithm should also be very close (if not identical) to the hash file built using the naive loading. Our further experiments prove this. In the experiment using the penalty score method to compare the query response time, we loaded 10 million records into a hash file using both our bulk loading algorithm and our naive loading (as discussed earlier). To make a fair query cost comparison, we forced the hash files in both cases to have the same load factors. We first generated a hash file using

our bulk loading algorithm, with parameters $E = 2$ and $O = 1$ and calculated the load factor. We then used the same load factor to generate another hash file for the same data set but using the naive loading algorithm. We randomly selected 100,000 queries from the data set and posed them to the two hash files. For each query, we measured both the number of I/Os and the running time. For linear hashing, a query ideally should cost one I/O, but the actual cost is typically a bit more due to the overflow records. As our measure of the number of I/Os, we count the number of buckets that are accessed for each query. This is reasonable since the queries are selected randomly and there cannot be much buffering effects. To measure the running time, we ran each query 100 times and calculated the average running time.

Table 5.7: Query cost comparison between the hash files (E/O)

	Size of the hash file	Load factor	Average query cost	Average I/O accesses
Bulk loading	1.37 GB	0.95	0.072 msec	1.10
Naive loading	1.37 GB	0.95	0.071 msec	1.10

As is shown in Table 5.7, using a penalty score function in our bulk loading method has the same average I/O accesses as the naive loading method. This result is the same as the user-specified average I/O method. The average query response times for the two hash files are also nearly the same.

We also did the same experiment as that of the the user-specified average I/O method on Berkeley DB. The parameters are set as: $E = 1$ and $O = 9$ which result in a load factor 0.47. The results are listed in Table 5.8.

Table 5.8: Query cost comparison with Berkeley DB (E/O)

	Size of the hash file	Load factor	Average query cost
Bulk loading	2.48 GB	0.47	0.070 msec
Berkeley DB	2.62 GB	0.47	4.80 msec

Chapter 6

Related Work

Closely related to our bulk loading is the incremental data organization of Jagadish et al. [10] which delays the insertions into a hash file. In this paper, they propose a "Stepped-Hash" algorithm, which 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. 6. Both methods reorder the input records to match the ordering of the hash file, hence reducing the number of random I/Os. A difference is that we use in-memory sorting and they use in-memory hashing. In the Stepped-Hash method, although records inside an in-memory hash bucket have the same value, there is no order at all inside a hash bucket. Records are mapped to random positions in the final hash file. In our methods, records are sorted based on their reversed hash values. Thus the records that are mapped to the same location in the hash file are all adjacent. In other words, the order of the records in a in-memory block is the same of the order of these records in the final hash file. This provides a slight benefit at the load time. In fact, for a part of the data set, partial sorting is always better than partial hashing, because records are more organized using sorting. An advantage of these two methods is that both of them support store data incrementally as it arrives.

However, a lot of splitting and record movements still occur when records are inserted into a non-empty hash table. In our experiment, 3 hours are used to load only 2 GB data using Berkeley DB.

Our Alg. 1 is more efficient and is different. The entire data is sorted in advance using external sorting which is typically fast. The total cost of loading is the cost of sorting, i.e. 3N if there is enough memory to store sqrt(N) pages, plus N for writing the hash file. The key point is we approximate the number of buckets in the final hash layout even before loading actually is performed. Therefore, there are no bucket splitting or record movements and buckets are not fetched again after they are written. On a data set with 20 million records, Alg. 1 is 50 times faster than our partial sort-based algorithm (Alg. 6) which is comparable to lazy insertions of Jagadish's Stepped-Hash algorithm.

Bercken et al. provide a generic approach to bulk loading multidimensional index structures [9]. Their method is based on an abstract data structure called buffer-tree. The buffer-tree differs from the target index structure mainly in two points. First, each internal node of the buffer-tree has an additional buffer where records are temporarily stored. Second, multiple insertions are processed simultaneously in the buffer-tree. When the number of records in the buffer exceeds a predefined threshold, the insertion processes of all records in the buffer advance to the next level of the tree. Standard routines for splitting and merging pages are used. In their method, sorting multidimensional data according to a predefined global ordering is avoided. Second, instead of inserting the record one by one, they insert multiple records simultaneously and multiple restructuring operations are also processed simultaneously in the tree. An example of how to apply this technique to R-tree was demonstrated in their paper. For bulk loading R-trees, their approach requires $O(n \log_m n)$ disk accesses in the worst case where n and m denote the number of data pages and available main memory (in pages), respectively.

Bercken's work avoid sorting the high dimensional data set with a predefined global ordering. However, this method is not easy to be applied to a hash index. Merging into a tree-based index is different from merging into a hash index. Two hash layouts may use different numbers of bits for addressing although their bit-randomizing hash functions are the same. Then merging them may cost a lot of bucket split and movements. To avoid them, the number of hash buckets or the number of bits for addressing has to be predicted before loading.

Böhm and Kriegel propose a generic bulk loading method which allows the application of user-defined split strategies in high dimensional index construction [3]. To determine the split dimension, they consider two cases: If the data subset fits in memory, the split dimension and subset size are obtained by computing selectivities or variances from the complete data subset. Otherwise, decisions are based on a sample of the subset which fits in memory and can be loaded without too many random disk accesses.

Fenk et al.'s work focus on bulk loading into a UB-Tree [6]. The UB-Tree is a multidimensional clustering index which inherits all good properties of a B-Tree. Logarithmic performance guarantees are given for the basic operations of insertion, deletion and point query. The UB-Tree clusters data according to the space filling Z-curve and proposes a new method to partition the data space into disjoint Z-regions. The Z-address which represents the position of a tuple in the Z-curve, determines the Z-region to which the tuple belongs. The key idea of Fenk's method is do an external sort of the data set according to their Z-addresses. Then B-tree standard techniques can be used when loading. This method can also be used for reorganizing UB-Trees and merging an existing UB-Tree with another UB-Tree or a new data set, because incremental loading only differs slightly from initial loading into a tree-based index. Both of them update existing pages.

Generally speaking, bulk loading into a tree-based or hash index can be classified into two groups:

- Algorithms using a kind of partition merge strategy, which partition the data set into small parts that fit to main memory. The index is constructed in memory on this part of the data set and then merged into the final layout. Jagadish's stepped-hash method, Bercken's bulk loading algorithm into multidimensional index belong to this group.

- Algorithms which apply an external sorting to reorder the data set and load the sorted data into the index. Our bulk loading algorithm (Alg. 1) and Fenk's bulk loading algorithm into a UB-Tree belong to this group.

Our partial algorithm (Alg. 6) is a combination of the above two strategies. We partition the data set into small parts which fit to main memory and sort the records in memory based on their reversed hash values.

Some bulk loading work has also been done in the area of Object oriented database. Wiener et al. study the problem of bulk loading into Object-oriented and object-relational databases (OODB) and propose their late-invsort method [18]. The problem is that the relationships among the objects make loading into an OODB rather slow. The data record of object A may show that there is a relationship between objects A and B. However, we can not assign an object ID to B when we read A because B may not have been read yet or it may have been read but is assigned a different ID. The inverse relationships (bidirectional relationships) exacerbate the problem. Inverse relationships are relationships that are maintained in both directions, so an update in one direction may cause a change to the other direction.

To solve the problem, Wiener et al. build a later updated inverse todo list to try to assign an OID when the data is read for the first time. Each todo list entry contains the OID of the object to be updated, the surrogate for the OID

to store in the object, and an Update offset at which to store the relationship. Surrogates that refer to objects described later in the data set are not assigned an OID immediately, but updated later. To avoid random reads and updates, they also sort the todo list so that the order of the entries corresponds to the creation order of the objects in the database. This pre-allocation of OIDs can avoid updates in the first place. In their performance study, they demonstrate that this could achieve an improvement of one to two orders of magnitude over the naive algorithm.

However, further experiments using larger data sets show that the performance of late-invsort algorithm degrades because the OIDs may be too large to fit in memory. To address this problem, Wiener et al. later provide a Partitioned-List Approach [19]. The key idea is that the id map is stored as a persistent partitioned list. If the id map is too large to fit in memory, they split the id map and the todo list into into several partitions such that each partition can fit in memory. Later, the todo list and inverse todo list are joined to create an update list. Finally, the update list are sorted by OID of the object to update and write them out in sorted runs. This algorithm has comparable performance to late-invsort algorithm for small data sets but does not degrade for large data sets. When the id map fits in memory, the partitioned list algorithm cost less than twice of that of the in-memory. When the id map does not fit in memory, in-memory method is inviable, partitioned-list method is at least an order of magnitude faster.

Not all of the commercial database systems support hash indexes. To the best of our knowledge, hash indexes are not supported in DB2, Sybase and Informix. Hash indexes are supported in Oracle, Microsoft SQL Server, PostgreSQL, MySQL and Berkeley DB (as discussed earlier), but we are not aware of any bulk loading algorithm for these indexes.

Chapter 7

Conclusions and Future Work

We have presented novel techniques for efficient bulk loading into a linear hash index. Our experiments confirm that our bulk loading algorithm improves upon alternative techniques, in terms of running time, by two orders of magnitude. We have shown how our proposed algorithm can be integrated into a commercial open source DBMS and have reported some of the improvements that can be obtained. Another contribution of this thesis is using a histogram to predict a hash layout before actually loading the index.

Using a histogram to predict the data distribution before loading opens up a few interesting research directions. The size of the memory that needs to be allocated varies with the skewness of the data; Using a fixed space to build the histogram, for instance, may not be sufficient for a highly skewed data set. At the same time, loading a highly skewed data is expected to waste the disk space and generally is not recommended. An alternative could be to transform the data (for instance using a different hash function) before loading. The trade-off between the allocated space for a histogram and the skewness of the data is not clear. Another issue is the type of the histogram that is being built. Instead of using a fixed-bucket width histogram, it might be better to use a multi-resolution approach that concentrates more histogram memory to regions of higher skew.

Our algorithms can be applied or extended when the hash file is not empty. For instance, our modified version of the *db_load* utility can still be used when the hash file is not empty. Our algorithms may also be applied when a hashing scheme other than linear hashing is used. For instance, our ordering of the records can be useful in an extendible hash file and can avoid many of the random accesses. Therefore, this method is still expected to improve the performance.

Bibliography

[1] http://www.postgresql-websource.com/psql736/source-dynahash.htm.

[2] Internet Archive. http://www.archive.org.

[3] C. Bohm and H. Kriegel. Efficient bulk loading of large high-dimensional indexes. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 251–260, 1999.

[4] A. Z. Broder, M. Najork, and J. L. Wiener. Efficient url catching for world wide web crawling. In *Proceedings of the International World Wide Web Conference (WWW)*, 2003.

[5] Berkeley DB. http://www.sleepycat.com.

[6] R. Fenk, A. Kawakami, V. Markl, R. Bayer, and S. Osaki. Bulk loading a data warehouse built upon a ub-tree. In *Proceedings of of IDEAS Conference*, pages 179–187, Yokohoma, Japan, 2000.

[7] J. Gray. A conversation with Jim Gray. *ACM Queue*, 1(4), 2003.

[8] G. R. Hjaltason, H. Samet, and Y. J. Sussmann. Speeding up bulk-loading of quadtrees. In *Proceedings of the International ACM Workshop on Advances in Geographic Information Systems*, pages 50–53, Las Vegas, November 1997.

[9] P. Widmayer J. Bercken, B. Seeger. A generic approach to bulk loading multidimensional index structures. In *Proc. of the VLDB Conference*, page 406, Athens, August 1997.

[10] H.V. Jagadish, P.P.S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *Proc. of the VLDB Conference*, pages 16–25, Athens, August 1997.

[11] W. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient resumption of interrupted warehouse loads. In *Proc. of the SIGMOD Conference*, pages 46–57, Dallas, May 2000.

[12] P. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, April 1988.

[13] Zipf's Law. http://www.nslij-genetics.org/wli/zipf.

[14] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the VLDB Conference*, pages 212–223, Montreal, October 1980.

[15] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University, 1981.

[16] S. Cluet S. Amer-Yahia. A declarative approach to optimize bulk loading into databases. *ACM Transactions on Database Systems*, 29(2):233–281, June 2004.

[17] M. Seltzer and O. Yigit. A new hashing package for unix. In *USENIX*, pages 173–184, Dallas, 1991.

[18] J. L. Wiener and J. F. Naughton. Bulk loading into an oodb: A performance study. In *Proceedings of the VLDB Conference*, pages 120–131, Santiago de Chile, Chile, September 1994.

[19] J. L. Wiener and J. F. Naughton. OODB bulk loading revisited: The partitioned-list approach. In *Proceedings of the VLDB Conference*, pages 30–41, Zurich, Switzerland, September 1995.