HASH-BASED INDEXING

The basic idea of hashing is to use a *hashing function*, which maps values in a search field into a range of *bucket numbers* to find the page on which a desired data entry belongs. The basic Static External Hashing scheme suffers from the problem of long overflow chains, which can affect performance.

Hash-based indexing techniques cannot support range searches. Tree-based indexing techniques can support range searches efficiently and are almost as good as hash-based indexing for equality selections. Thus many commercial systems choose to support only tree-based indexes. Nonetheless, hashing techniques prove to be very useful in implementing relational operations such as joins.

1. STATIC HASHING

The pages containing the data can be viewed as collection of **buckets**, with one **primary** page and possibly additional **overflow** pages per bucket. A file consists of buckets 0 through N-1, with one primary page per bucket initially.

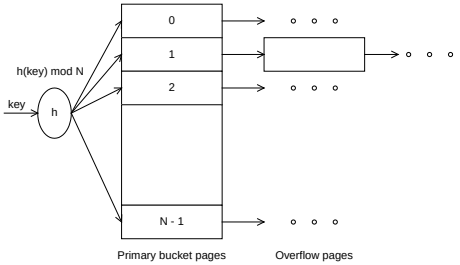

Figure 6.1 Static Hashing

- To search for a data entry, we apply a **hash function h** to identify the bucket to which it belongs and then search this bucket. To speed the search of a bucket, we can maintain data entries in sorted order by search key value.
- In order to insert a data entry, we use the hash function to identify the correct bucket and then put the data entry there. If there is no space for this data entry, we allocate a new *overflow page*, put the data entry on this page, and add this page to **overflow chain** of the bucket.
- To delete a data entry, we use the hashing function to identify the correct bucket, locate the data entry by searching the bucket, and then remove it. If this data entry is the last in a overflow page, the overflow page is removed from the overflow chain of the bucket and added to a list of *free pages*.

- Since the number of buckets in a static hashed file is known when the file is created, the primary pages can be stored on successive disk pages. Thus a search ideally requires just one disk I/O, and insert and delete operations require two I/Os (read and write the page), although the cost could be higher in the presence of overflow pages.
- As the file grows, long overflow chains can develop. Since searching a bucket requires searching all pages in its overflow chain, performance deteriorates.
- The main problem with Static Hashing is that the number of buckets is fixed. If a file shrinks greatly, a lot of space is wasted, if a file grows a lot, long overflow chains develop, resulting in poor performance. One alternative is to periodically 'rehash' the file to restore the ideal situation (no overflow chains, about 80% occupancy). However, rehashing takes time and the index cannot be used while rehashing is in progress.

2. EXTENDIBLE HASHING

Idea: to overcome the problem with inserting a new data entry into a full bucket, use a **directory** of pointers to buckets, and double the size of number of buckets by doubling just the directory and splitting *only* the bucket that overflowed.

Example:

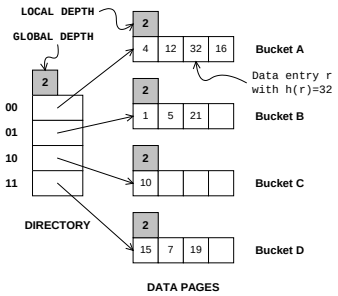

Figure 6.2 Example of an Extendible Hashed File

The directory consists of an array of size 4, with each element being a pointer to a bucket. To locate a data entry, we apply a hash function to the search field and **take the last two bits of its binary representation** to get a number between 0 and 3. The pointer in this array position gives us the desired bucket; we assume that each bucket can hold four data entries. Thus to locate a data entry (e.g. 13) with hash value 5 (binary 101), we look at directory element 01 and follow the pointer to the data page (bucket B).

Let us consider insertion of a data entry into a full bucket. Consider the insertion of data entry 20 (binary 10100). Looking at directory element 00, we are led to bucket A, which is already full. We must first **split** the bucket by allocating a new bucket and redistributing the contents (including the new entry to be inserted) across the old bucket and its 'split image'. To redistribute entries across the old bucket and its split image, we consider the last *three* bits of *h(r)*; the last two bits are 00, indicating a data entry that belongs to one of these two buckets, and the third bit discriminates between these buckets. The redistribution of entries is illustrated in Figure 4.

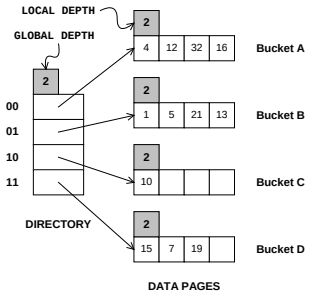

Figure 6.3 After Inserting Entry *r* with *h(r)*=13

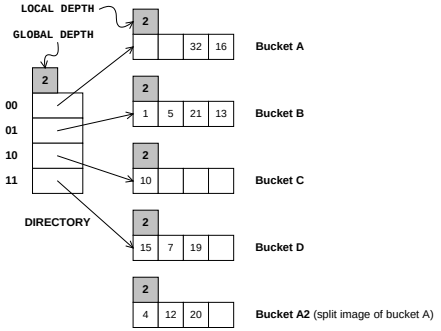

Figure 6.4 While Inserting Entry *r* with *h(r)*=20

Notice a problem that we must now resolve - we need three bits to discriminate between two of our data pages (A and A2), but the directory has only enough slots to store all two-bit patterns.

The solution is to *double the directory*. Elements that differ only in the third bit from the end are said to 'correspond': *corresponding elements* of the directory point to the same bucket with the exception of the elements corresponding to the split bucket. In our example, bucket 0 was split so, new directory element 000 points to one of the split versions and element 100 points to the other. The sample file after completing all steps in the insertion of 20 is shown in Figure 5.

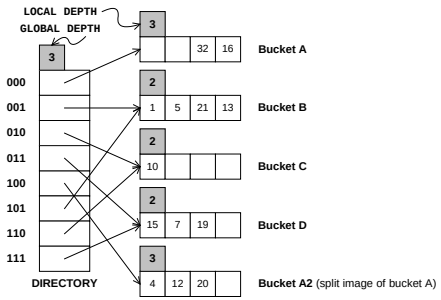

Figure 6.5 After Inserting Entry *r* with *h(r)*=20

We observe that the basic technique used in Extendible Hashing is to treat the result of applying a hash function *h* as a binary number and to interpret the last *d* bits, where *d* depends on the size of directory, as an offset into the directory. The number *d* is called the **global depth** of the hashed file and is kept as part of the header of the file. It is used every time we need to locate a data entry.

Question: whether splitting a bucket necessitates a directory doubling?

We now insert 9, it belongs in bucket B; this bucket is already full. We can deal with this situation by splitting the bucket and using directory elements 001 and 101 to point to the bucket and its split image, as shown in Figure 6.

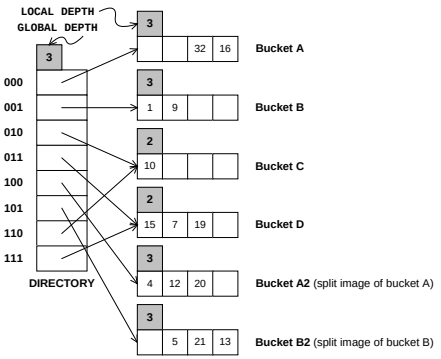

Figure 6.6 After Inserting Entry *r* with *h(r)*=9

In order to differentiate between these cases, and determine whether a directory doubling is needed, we maintain a **local depth** for each bucket. If a bucket whose local depth is equal to the global depth is split, the directory must be doubled.

Initially, all local depths are equal to the global depth (which is the number of bits needed to express the total number of buckets). We increment the global depth by 1 each time the directory doubles. Also, whenever a bucket is split (whether or not the split leads to a directory doubling), we increment by 1 the local depth of the split bucket and assign this same (incremented) local depth to its (newly created) split image. Intuitively, if a bucket has local depth l, the hash values of data entries in it agree upon the last l bits; further, no data entry in any other bucket of the file has hash value with the same last l bits. A total of 2^{d-l} directory elements point to a bucket with local depth l; if $d = l$, exactly one directory element is pointing to the bucket, and splitting such a bucket requires directory doubling.

A final point to note is that we can also use the first d bits (the *most significant* bits) instead of the last d (*least significant* bits), but in practice the *last d* bits are used. The reason is that a directory can be doubled simply by coping it.

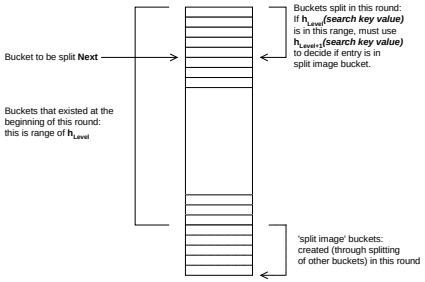

Figure 6.7 Buckets during a Round in Linear Hashing

Consider how we search for a data entry with a given search key value. We apply hash function h_{level}, and if this leads us to one of the unsplit buckets, we simply look there. If it leads us to one of the split buckets, the entry may be there or it may have been moved to the new bucket created earlier in this round by splitting this bucket; to determine which of these two buckets contains the entry, we apply $h_{level+1}$.

For deletes, the data entry is located and removed. If the delete leaves the bucket empty, it can be merged with its split image, although this step is often omitted in practice. Merging buckets decreases the local depth. If each directory entry element points to the same bucket as its split image, we can halve the directory and reduce the global depth, although this step is not necessary for correctness.

On the other hand, the directory grows in spurts and can become large for *skewed data distributions* (where our assumption that data pages contain roughly equal numbers of data entries is not valid). In the context of hashed files, a **skewed data distribution** is one in which the distribution of *hashed-values of search field values* (rather than the distribution of search field values themselves) is skewed (very 'bursty' or not uniform). Even if the distribution of search values is skewed, the choice of a good hashing function typically yields a fairly uniform distribution of hash-values; skew is therefore not a problem in practice.

We now describe Linear Hashing in more detail. A counter *Level* is used to indicate the current round number and is initialized to 0. The bucket to split is denoted by *Next* and is initially bucket 0 (the first bucket). We denote the number of buckets in the file at the beginning of round *Level* by N_{Level}. We can easily verify that $N_{Level} = N * 2^{Level}$. Let the number of buckets at the beginning of round 0, denoted by N_0, be N. We show a small linear hashed file in Figure 6.8. Each bucket can hold four data entries, and the file initially contains four buckets, as shown in the figure.

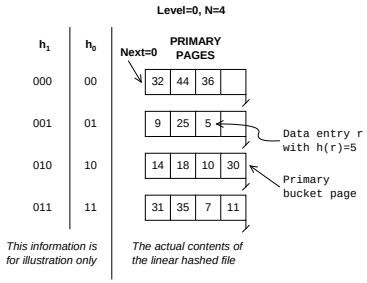

Figure 6.8 Example of a Linear Hashed File

3. LINEAR HASHING

The scheme utilizes a family of hash functions h_0, h_1, h_2, ... , with the property that each function's range is twice that of its predecessor. That is, if h_i maps a data entry into one of M buckets, h_{i+1} maps a data entry into $2M$ buckets. Such a family is typically obtained by choosing a hash function h and an initial number N of buckets, and defining $h_i(value) = h(value) \bmod (2^i N)$. If N is chosen to be a power of 2, then we apply h and look at the last d_i bits; d_0 is the number of bits needed to represent N, and $d_i = d_0 + i$. Typically we choose h to be a function that maps a data entry to some integer.

The idea is best understood in terms of **rounds** of splitting. During round number *Level*, only hash functions h_{level} and $h_{level+1}$ are in use. The buckets in the file at the beginning of the round are split, one by one from the first to the last bucket, thereby doubling the number of buckets. At any given point within a round, therefore, we have buckets that have been split, buckets that are yet to be split, and buckets created by splits in this round, as illustrated in Figure 6.7.

We have considerable flexibility in how to trigger a split, thanks to the use of overflow pages. We can split whenever a new overflow page is added, or we can impose additional conditions based on conditions such as space utilization. For our examples, a split is 'triggered' when inserting a new data entry causes the creation of an overflow page.

Whenever a split is triggered the *Next* bucket is split, and hash function $h_{Level+1}$ redistributes entries between this bucket (say bucket number b) and its split image; the split image is therefore bucket number $b + N_{Level}$. After splitting a bucket, the value of *Next* is incremented by 1. In this example file, insertion of data entry 43 triggers a split. The file after completing the insertion is shown in Figure 6.9.

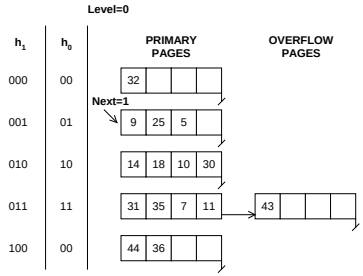

Figure 6.9 After Inserting Record *r* with *h(r)*=43

At any time in the middle of round *Level*, all buckets above bucket *Next* have been split, and the file contains buckets that are their split images, as illustrated in Figure 7. Buckets *Next* through N_{Level} have not yet been split. If we use h_{Level} on a data entry and obtain a number *b* in the range *Next* through N_{Level}, the data entry belongs to bucket *b*. For example, $h_0(18)$ is 2; since this value is between current values of *Next* (=1) and N_1 (=4), this bucket has not been split.

However, if we obtain a number *b* in range 0 through *Next*, the data entry may be in this bucket or in its split image (which is bucket number $b + N_{Level}$); we have to use $h_{Level+1}$ to determine which of these two buckets the data entry belongs to.

For example, $h_0(32)$ and $h_0(44)$ are both 0. Since *Next* is currently equal 1, which indicates a bucket that has been split, we have to apply h_1. We have $h_1(32) = 0$ and $h_1(44) = 4$. Thus 32 belongs in bucket A and 44 belongs in its split image, bucket A2.

Not all insertions trigger a split. If we insert 37 into the file shown in Figure 6.9, the appropriate bucket has space for the new data entry. The file after the insertion is shown in Figure 6.10.

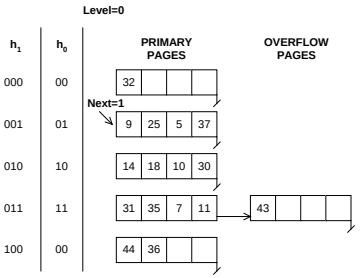

Figure 6.10 After Inserting Record *r* with *h(r)*=37

Sometimes, the bucket pointed to by *Next* (the current candidate for splitting) is full, and a new data entry should be inserted in this bucket. In this case a split is triggered, but we do not need a new overflow bucket.

This situation is illustrated by inserting 29 into file shown in Figure 6.10. The result is shown in Figure 6.11.

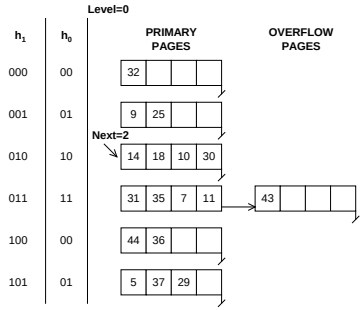

Figure 6.11 After Inserting Record *r* with *h(r)*=29

When *Next* is equal to N_{Level} - 1 and a split is triggered, we split the last of the buckets that were present in the file at the beginning of the round *Level*. The number of buckets after the split is twice the number at the beginning of the round, and we start a new round with *Level* incremented by 1 and *Next* reset to 0. Consider the example file in Figure 6.12, which was obtained from the file of Figure 11 by inserting 22, 66 and 34. Inserting 50 causes a split that leads to incrementing *Level* (see Figure 6.13).

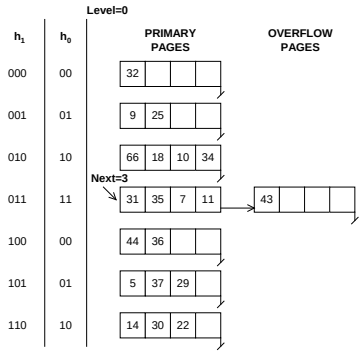

Figure 6.12 After Inserting Records with *h(r)*=22, 66 and 34

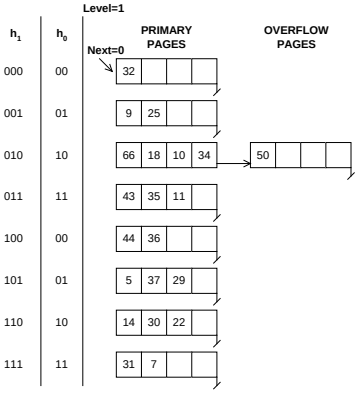

Figure 6.13 After Inserting Record *r* with *h(r)*=50

In summary, an equality selection costs just one disk I/O unless the bucket has overflow pages; in practice the cost on average is about 1.2 disk accesses for reasonably uniform data distributions. The cost can be considerably worse - linear in the number of data entries in the file - if the distribution is very skewed. Inserts require reading and writing a single page, unless a split is triggered.

The deletion is essentially the inverse of insertion. If the last bucket in the file is empty, it can be removed and *Next* can be decremented. If we wish, we can combine the last bucket with its split image even when it is not empty, using some criterion to trigger this merging, in essentially the same way. The criterion is typically based on the occupancy of the file, and merging can be done to improve space utilization.

4. EXTENDIBLE HASHING VERSUS LINEAR HASHING

We observe that the choice of hashing function is actually very similar to what goes on in Extendible Hashing - in effect, moving from h_i to h_{i+1} in Linear Hashing corresponds to doubling the directory in Extendible Hashing. Both operations double the effective range into which key values are hashed; but whereas the directory is doubled in a single step, moving form h_i to h_{i+1}, along with a corresponding doubling in the number of buckets, occurs gradually over the course of a round. The new idea behind Linear Hashing is that a directory can be avoided by clever choice of the bucket to split. On the other hand, by always splitting the appropriate bucket, Extendible Hashing may lead to a reduced number of splits and higher bucket occupancy.

The disadvantage of Linear Hashing relative to Extendible Hashing is that space utilization could be lower, especially for skewed distributions, because the bucket splits are not concentrated where the data density is highest, as they are in Extendible Hashing. A directory-based implementation can improve space occupancy, but it is still likely to be inferior to Extendible Hashing in extreme cases. We can address this problem by adjusting the criterion used to trigger splits; in effect, we can trade off slightly longer overflow chains for better space utilization.