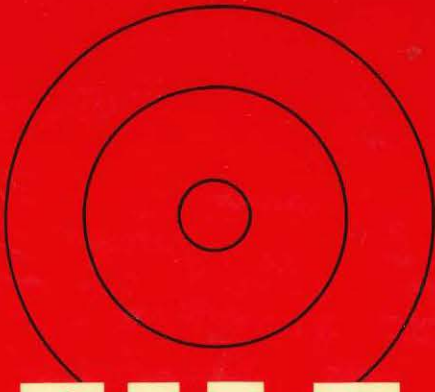
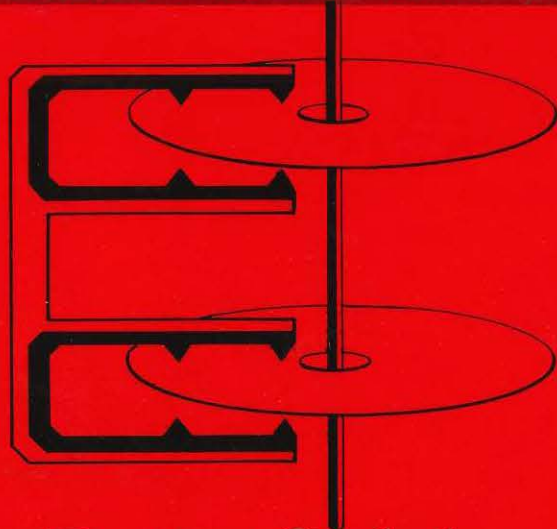




FILE STRUCTURES



AN ANALYTIC APPROACH



Betty Salzberg

Symbol Table

notation	definition	section
b	number of blocks in a file	2.3
B	number of bytes in a block	2.3
Bfr	number of records in a block	2.4
bk	number of buckets in a file	2.4
$Bkfr$	number of records in a bucket	5.2
btt	block transfer time (ms)	2.3
C	number of blocks per cylinder	2.3
dt	data transfer time for a bucket (ms)	2.4
ebt	effective block transfer time (ms)	2.3
fo	fan-out	5.2
Lf	load factor	6.1
m	time to move to adjacent cylinder (ms)	2.3
M	number of buckets in primary area	6.1
mem	number of blocks in memory	5.2
n	number of records in a file	2.4
N	number of cylinders in a disk	2.3
nsg	number of original segments	4.2
P	for P -way merge	4.2
r	average rotational latency (ms)	2.3
R	number of bytes in a record	2.4
s	average seek time (ms)	2.2
seg	number of blocks in segment	4.2
t	speed of disk drive (bytes/ms)	2.3
t'	formatted speed (bytes/ms)	2.3
T_D	time to delete one record	3.1
T_F	time to fetch one record	3.1
T_I	time to insert one record	3.1
T_N	time to fetch next record	3.1
T_U	time to update one record	3.1
T_X	time to do exhaustive read	3.1
T_Y	time to reorganize file	3.1
v	minimum number of keys in B^+ -tree node	5.2

FILE STRUCTURES: AN ANALYTIC APPROACH

BETTY SALZBERG
Northeastern University



PRENTICE HALL
Englewood Cliffs, N.J. 07632

Library of Congress Cataloging-in-Publication Data

SALZBERG, BETTY JOAN.
File structures.

Bibliography: p.

Includes index.

1. File organization (Computer science) I. Title.
QA76.9.F5S25 1987 005.74 87-25747
ISBN 0-13-314691-X

Editorial/production supervision: **Ellen B. Greenberg**
Cover design: **Photo Plus Art**
Manufacturing buyer: **Cindy Grant**

To my parents, Ted and Hilda Salzberg
and to Lawrence and Pearl



© 1988 by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

All rights reserved. No part of this book may be
reproduced, in any form or by any means,
without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-314691-X 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Preface	v
Acknowledgments	ix
1 Introduction	1
1.1 Large Dynamic Files	1
1.2 Operations on Files	3
1.3 How Estimates are Made	5
1.4 Mathematical Preliminaries	7
2 Hardware	10
2.1 Disks	10
2.2 Seek Time	16
2.3 Rotational Latency and Block Transfer Time	19
2.4 Records and Buckets	26
2.5 Double Buffering	35
Appendix: Calculating Seek Time	40
Chapter Summary	43
3 Sequential Files	48
3.1 Unordered Sequential Files (pile files)	48
3.2 Sorted Sequential Files	68
Chapter Summary	85

4	External Sorting	94
4.1	Creating Initial Sorted Segments	94
4.2	One Disk Drive Large-Memory Merging	107
4.3	Two Disk Drive Large-Memory Merging	116
	Appendix: Choosing the Number of Passes	127
	Chapter Summary	129
5	Tree Indexes	139
5.1	ISAM	139
5.2	B ⁺ -Trees	143
5.3	Range Queries and Leaf Size	158
5.4	Insertion and Deletion in B ⁺ -Trees	170
5.5	Constructing a B ⁺ -Tree for an Existing File	184
	Chapter Summary	189
6	Hashing	202
6.1	Hashing with Buckets and Chaining	202
6.2	Linear Hashing	217
6.3	Constructing a Hash Table	228
	Appendix: Bounded Index Extendible Hashing	234
	Chapter Summary	241
7	Managing Several Record Types	252
7.1	Database Management Systems	252
7.2	Multilists	257
7.3	Natural-Join Queries	266
	Appendix: The System R Query Optimizer	286
	Chapter Summary	294
8	Special Topics	304
8.1	Introduction	304
8.2	Bounded Disorder Files	305
8.3	The Grid File	312
8.4	The hB-Tree	319
8.5	Optical Disks	325
8.6	Concurrency and Recovery	330
	References	338
	Index	341

6

Hashing

6.1 HASHING WITH BUCKETS AND CHAINING

Suppose that we knew that the keys of the records of some file were the numbers from 1 to 1000. Then we could store the record with key 1 in position 1 on the disk, the record with key 2 in position 2, and so on. Then to find a record with key x , we could go directly to position x . The value of the key would yield the address of the key. This would be faster than using a B^+ -tree for individual record searches. This would be a one-disk-access method.

Hashing is a generalization of this process. A *hashing algorithm* or *hash function* is a means of calculating the disk address of a block containing a given record from the value of its key. Sometimes hashing is called *direct access* or *random access* or *key-to-address transformation*.

Buckets

We set up an area to keep the records. We look for an area that will hold the number of records we expect to have, with a little room left over. This is called the *primary area*. We divide the primary area into *buckets*. As usual, a bucket may consist of one or several blocks.

Suppose that there are M buckets in our primary area. We then calculate a number from the key. For example, if the key is alphanumeric, we can add up all the ASCII numbers of each character in the key. This gives us a large number. Then we divide the number by M and take the remainder. This will give us a number between 0 and $M - 1$, which will correspond to one of the buckets in our storage area. Most of the time this kind of calculation will produce numbers which more or less evenly distribute the key values among the different bucket labels.

Note that hash functions *never* preserve order. The point of hash functions is to take a highly nonuniform distribution—probably one with data unevenly clustered—and transform it into one which is nearly uniform. This cannot be done while preserving order. It is *always* a bad idea to use field values themselves for hashing. Real data is never uniformly distributed.

We place the record whose key calculation is a number x into the bucket numbered x . More than one key may give the same number. When the hash function yields the same number for two different keys we say we have a *collision*. This does not cause a problem as long as there is still room in the bucket with that number.

Separate Chaining

When the bucket fills up and another record is entered whose hash value is the same as the records in the full bucket, we add the address of a new bucket in a special area called the *overflow area*. The overflow area is disjoint from the primary area. We usually keep some overflow area in each cylinder.

Adding the address of an overflow bucket to a primary-area bucket which is full is called *chaining*. Since there is a separate chain of overflow buckets for each primary-area bucket, this is sometimes called *separate chaining*. This is the usual method used for hashing collision resolution on disks. First, all records with the same hash value are placed in the same bucket. When that bucket overflows, a link or address is added referring to an overflow bucket. This collision-resolution method is called *hashing with buckets and separate chaining*. Almost all existing file systems and database management systems use this method of hashing if any hashing is used at all.

Hashing with buckets and separate chaining is illustrated in figure 6.1. In this illustration, we have five buckets in the primary area. The hash function, $h(x)$, is $x \bmod 5$. In our example, the bucket factor, $Bkfr$, or the number of records in a bucket, is three. After three records are inserted whose key has the same remainder mod 5, a chain to a bucket in the overflow area is constructed.

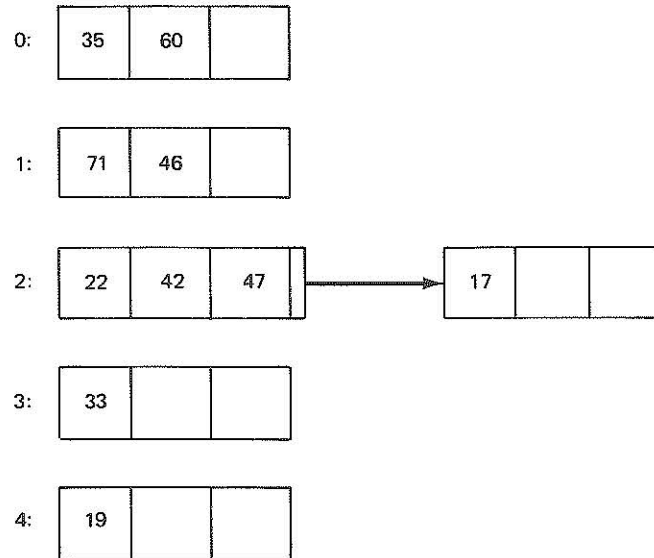


Figure 6.1. Hashing with buckets and separate chaining. Here the bucket factor is 3, and the hash function is $h(x) = x \bmod 5$.

The Load Factor

If we allocate exactly the amount of space needed for the file to the primary area, there will be many chains. The hash functions can be expected to more or less randomly distribute the records, but there will be a few more records with one hash value than with another. To limit the amount of overflow, we allocate more space to the primary area than we need. We want to allocate space so that the primary area will be only part full, say 70% full or 90% full.

We calculate what percentage we would like and allocate space depending on how much data we expect. The number of records in the file divided by the number of places for records in the primary area is called the *load factor*, Lf . So we try to allocate space for an expected load factor.

As usual, $Bkfr$ stands for the number of records which one bucket can hold. We recall that M stands for the number of buckets in the primary area. Then

$$Lf = n / (M \times Bkfr)$$

Here n is the number of records in the file as usual. $M \times Bkfr$ is the number of places for records in the primary area. Let us investigate how a choice of load factor and of bucket factor can affect the time for fetching a record.

Fetching Using Buckets and Chains

If the distribution of values is even and the original area chosen is big enough, hashing is very efficient for fetching. The time can be near one disk access. That is, as long as there is no overflow,

$$T_F = s + r + dtt$$

We use dtt as we did for large leaves in B^+ -trees. A bucket may consist of several blocks, and dtt is the time to transfer the data in one bucket.

In figure 6.2, we see a chart of how many accesses are needed for a given load factor and a given bucket factor. This is from Knuth (1973). In general, a smaller load factor means less overflow, and so a faster fetch time. However, a smaller load factor means more wasted space. If the load factor is 50%, the fetch time will be excellent, but half the space on the disk will be wasted. This is a well-understood trade-off in file design.

A larger bucket factor also means less overflow in general, at least for load factors up to 90%. However, a larger bucket factor means a larger dtt , so that each access to a bucket takes longer. That is, the number of accesses for larger buckets and a given load factor is less than the number of accesses for smaller buckets. But each access takes more time, since the data transfer time is larger.

In addition, there are choices we can make for the way overflow is handled. We shall use the simplest model in our analysis: bucket chains and overflow buckets the same size as the primary-area buckets. However, other choices could have been made. We could have decided to make the overflow buckets smaller than the primary-area buckets. We could have decided to have several primary-area buckets share the same overflow buckets and make chains between records with the same hash value. In fact, Knuth's calculations in figure 6.2 assume record chains, not bucket chains. Our assumption of the use of bucket chains will make space utilization worse, since most overflow buckets will be only partially full. The time estimates are approximately the same. ✓

Table A
AVERAGE ACCESSES IN AN *UNSUCCESSFUL* SEARCH BY SEPARATE CHAINING

Bucket size, <i>Bk fr</i>	Load factor, <i>Lf</i>									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0048	1.0187	1.0408	1.0703	1.1065	1.1488	1.197	1.249	1.307	1.3
2	1.0012	1.0088	1.0269	1.0571	1.1036	1.1638	1.238	1.327	1.428	1.5
3	1.0003	1.0038	1.0162	1.0433	1.0898	1.1588	1.252	1.369	1.509	1.6
4	1.0001	1.0016	1.0095	1.0314	1.0751	1.1476	1.253	1.394	1.571	1.7
5	1.0000	1.0007	1.0056	1.0225	1.0619	1.1346	1.249	1.410	1.620	1.7
10	1.0000	1.0000	1.0004	1.0041	1.0222	1.0773	1.201	1.426	1.773	2.0
20	1.0000	1.0000	1.0000	1.0001	1.0028	1.0234	1.113	1.367	1.898	2.3
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0007	1.018	1.182	1.920	2.7

Table B
AVERAGE ACCESSES IN A *SUCCESSFUL* SEARCH BY SEPARATE CHAINING

Bucket size, <i>Bk fr</i>	Load factor, <i>Lf</i>									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0500	1.1000	1.1500	1.2000	1.2500	1.3000	1.350	1.400	1.450	1.5
2	1.0063	1.0242	1.0520	1.0883	1.1321	1.1823	1.238	1.299	1.364	1.4
3	1.0010	1.0071	1.0216	1.0458	1.0806	1.1259	1.181	1.246	1.310	1.4
4	1.0002	1.0023	1.0097	1.0257	1.0527	1.0922	1.145	1.211	1.290	1.3
5	1.0000	1.0008	1.0046	1.0151	1.0358	1.0699	1.119	1.186	1.286	1.3
10	1.0000	1.0000	1.0002	1.0015	1.0070	1.0226	1.056	1.115	1.206	1.3
20	1.0000	1.0000	1.0000	1.0000	1.0005	1.0038	1.018	1.059	1.150	1.2
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.015	1.083	1.2

Figure 6.2. The number of disk accesses for a given load factor and a given bucket factor (from Knuth 1973).

Thus, performance can be enhanced by the choice of bucket size and of load factor. Space utilization can be enhanced by the choice of load factor and by the choice of a policy for handling overflow. This is a typical file-design problem. If the file does not grow too much, hashing can be fine-tuned to give excellent performance and excellent space utilization.

Database Growth

Even good choices for load factors and overflow policies will not give excellent performance when the database grows too large. When too many records are added, the overflow chains become long, just as with ISAM. Most file systems which use hashing have reorganization utilities to read out a hashed file and write it back to the disk in a larger table.

If there has been overflow and there is an average chain length of x ,

$$T_F(\text{successful}) = \overbrace{s + r + dtt} + (x/2) \times \overbrace{(s + r + dtt)}$$

This is one access to get the first record in the chain (the bucket in the primary area) and a traversal of about half the average chain length. That is, in any case, we go first to the bucket in the primary area. Then if the record is not there, we check the first overflow bucket. We continue until we find the record. If the record is there, on average we shall find it when we have looked at half the buckets in the chain.

An attempt to fetch a record which is not in the file (an *unsuccessful search*) will take more time on average than a *successful search*. This is due to the fact that the whole chain must be examined to determine that a record is not there; an unsuccessful search can never quit after reading only part of the chain. That is, for an unsuccessful search we have

$$T_F(\text{unsuccessful}) = s + r + dtt + x \times (s + r + dtt)$$

Note that we did not need two cases for the B^+ -tree. The number of accesses is the same for both successful and unsuccessful searches in a B^+ -tree.

If there has been a great deal of overflow in a hash table, an unsuccessful search will take much longer on average than a successful one. Without much overflow, both successful and unsuccessful searches take about one disk access.

For example, if $x = 10$, the number of disk accesses for a successful search is about 6, counting the first access, which must always be made. For an unsuccessful search, we must make 11 disk accesses, 1 to the primary area and 10 to the overflow area. But if $x = 1/10$, the number of disk accesses is about 1, whether or not the search is successful.

This illustrates both the power and the limitations of hashing with buckets and chains. If the file designer estimates the size of the

database correctly, hashing can be the fastest access method available. If not, the performance can degrade considerably as more records are added than were anticipated. When this happens the file must be unloaded and reloaded into a new hash structure. This is how most hashing schemes in existence today behave. They require frequent reorganization as more and more data is added.

Deletion

There are two very good ways to handle deletion. One model of deletion does not include any consolidation of sparse buckets. This is in fact how many hashing schemes work. A record to be deleted is merely marked with a tombstone, as in the pile file and the sorted files in chapter 3. When the whole hash table is reorganized because of declining performance, these records are not copied to the new table.

Another model, the one we choose to use in this book, has us consolidate when deleting. When we delete a record, we replace it with the last record in the chain of the current bucket. This makes deletion a more complex operation than it would be if we were using tombstones.

In addition, we deallocate the last bucket in a chain of buckets when it becomes empty from successive deletions. That is, on deleting, we move the last record with the same hash value to the place of the deleted record and deallocate any empty overflow buckets.

This makes searching more efficient. But we have to do something very odd for deletion. We have to bring in a whole chain to memory. To do this, we have to perform an *unsuccessful* fetch, even though we are successful in finding the record to be deleted. We bring in the whole chain, and we move the last record to the place of the deleted one. We have to modify the bucket with the deleted record and the bucket from which the last record is moved. They may be the same bucket.

We shall use the following formula for deletion:

$$T_D = T_F (\text{unsuccessful}) + 2r$$

This assumes that most of the time, the record is removed from the last bucket. We do not count the cost of modifying another bucket or of removing the last bucket from a chain of buckets. The point is that we bring in the chain to memory, and we modify one or two buckets. Deletion is a relatively inexpensive incremental operation.

Most hashing schemes in use today do not consolidate space when deleting records. Since they have to be reorganized often, consolidation takes place during the reorganization phase. The old hash table with tombstones is read in, and the new hash table without tombstones is written to the disk. We shall show how to do reorganization efficiently in section 6.3.

However, new methods such as those we shall consider in section 6.2 do not become reorganized. Thus, it is more important to keep things cleaned up. To be somewhat consistent, we choose to use a deletion method with consolidation in all cases. That is, we have chosen our deletion method for pedagogical reasons only. This is the way we predict that deletion will be handled in the future, not the way that it is handled now. Current file systems use tombstones.

Insertion

For insertion, we assume that new records are inserted at the end of the chain. Thus, when we do not have to add a new bucket to the chain, insertion means modifying the last bucket. The time for insertion is

$$T_I = T_F (\text{unsuccessful}) + 2r$$

We have not considered the cost for adding an extra bucket when the last bucket is full. The cost of inserting a record is essentially a fetch and a rotation of the disk for modifying the last bucket in a chain. The point is that this is pretty efficient.

Sequential Operations

Hashing is useless for sequential operations such as range searches or listing a file in order, since hash functions are *never* order preserving. For example, suppose that the California tax records are organized by a hash table based on social security number. This means that there is a randomizing function calculated from the social security number and a remainder taken from this number. The records are *not* stored in order of social security number. Two records whose social security numbers differ only in the last digit may be nowhere near each other.

If we want to read the records of all the California taxpayers whose social security number is between 000-00-0000 and 000-99-9999 from the file of California tax records, we either have to make 1 million separate fetches for the 1 million different numbers between 0 and

999,999, many of which will not correspond to records in the file, or read sequentially, ignoring the hash structure. In this case we at least know all the values in the range. Making one million hash-table fetches will take about 7 hours. Reading sequentially will take 20 minutes, accounting for 30% wasted space in the hash table.

Suppose that we had an alphabetic field like LAST NAME. If we want all the taxpayers between “Jones” and “Smith,” what values must we hash? We cannot use the hash structure in this case; we have to read sequentially. That is, we must read the file from beginning to end and pick out the records which qualify. The hash structure, which calculates a number from an exact key value, cannot be used for this.

Since the hash function is not order preserving, the time for reading records in order using a hash table is

$$T_X = n \times T_F$$

This presumes that we have a list of key values in order. This would be an inefficient method for reading in order; it is better to sort first.

A Mix of Operations

Suppose that we have the California income tax file of 6 million 400-byte records. Every month, we make 10,000 random individual-record fetches, and we have ten small-range queries, each for about 3% of the file. We want to choose between the following organizations:

1. Use a hash table for the individual-record fetches, and read sequentially through the file for the range queries. Suppose that the hash table has a load factor of 70% and a bucket factor of 50, so that each fetch is about one disk access.
2. Use the same hash table for the individual-record fetches, but keep a secondary B⁺-tree index for the small-range queries. Suppose that the secondary B⁺-tree index has not just bucket addresses for the records but *block* addresses for each record. Suppose that all the nodes in the secondary B⁺-tree have 2400 bytes.
3. Use a primary B⁺-tree for both the individual-record fetches and the small-range searches. Assume that the B⁺-tree leaf-node size has been chosen to make this mix of operations optimal. Assume that the internal nodes are 2400 bytes and that we have a two-disk-access method.

We know that the B^+ -tree will be less efficient than hashing for the individual record fetches. But what will be best for this mix of operations?

First, let us calculate the time it takes to do 10,000 individual-record fetches using hashing. Since we have a large bucket factor and a small load factor, the number of disk accesses for one fetch is about one. However, the large bucket factor and the 400-byte records imply that the data transfer time for one bucket is fairly large. The size of the bucket is 21,600 bytes. It had to be a multiple of 2400 bytes. This is nine blocks. The amount of time to read 21,600 bytes is about 7.5 milliseconds. (We divided 21,600 by the effective data transfer rate, 2857 bytes per millisecond.) Thus, the time for 10,000 fetches using the hash table is

$$10,000 \times (16 + 8.3 + 7.5) \text{ ms} = 5.3 \text{ min}$$

That is, it takes only about 32 milliseconds to fetch the correct bucket from a file of 6 million records using a hash table. When we do this 10,000 times, it takes us about 5.3 minutes.

Now, the time to read the whole California file was 14 minutes when it was stored as a pile file without any structure. But the load factor of 70% implies that 30% of the space here is wasted. As we saw in chapter 2, sequential reading is slower when there is wasted space. This also happens when we have a B^+ -tree structure where only $\ln 2$ of the space is used. Thus, the time for a sequential read through the California file which has a hash-table structure with a load factor of 70% is $14/(70\%)$ or 20 minutes. To do ten range searches using sequential reading, it takes us 200 minutes.

If we had a secondary B^+ -tree set up to do the range searches, the time would be

$$10 \times 0.03 \times n \times (s + r + btt)$$

This is ten range searches, each for 0.03 of the file, using the leaves of the secondary B^+ -tree for the addresses of the records in the range. We use btt instead of dt , since we have stipulated that the B^+ -tree have the block addresses and not just the bucket addresses. Our calculation becomes

$$10 \times 0.03 \times 6,000,000 \times 25.1 \text{ ms}$$

This is 758 minutes. So even for range searches of 3% of this large file, the B⁺-tree is not as efficient as sequential reading.

So far, it looks as if we should use hashing for the individual-record searches and sequential reading for the range queries.

Now let us see just how well we can do with a primary B⁺-tree set up for this mix of operations. The formula we want to minimize is

$$\begin{aligned} 10,000 \times (2r + 2s + btt + dtt) + 10 \times 0.03 \times bk \times (r + s + dtt) \\ = 10,000 \times (49.4 + dtt) + 0.3 \times bk \times (24.3 + dtt) \end{aligned}$$

Here, $dtt = x/2857$, where x is the number of bytes in the bucket. We also have

$$bk = \frac{n}{Bkfr \times \ln 2} = \frac{6,000,000}{(x/400) \times \ln 2} = \frac{3.5 \times 10^9}{x}$$

So we have

$$\begin{aligned} 10,000 \times (49.4 + x/2857) + (0.3 \times 3.5 \times 10^9/x) \times (24.3 + x/2857) \\ = \text{constant} + (10,000/2857) \times x + 25.5 \times 10^9/x \end{aligned}$$

Taking the derivative with respect to x and setting it equal to 0, we obtain x equal to about 86,400 bytes, or about 36 blocks. Then dtt is about 30.2 milliseconds. The time for 10,000 individual-record fetches is

$$10,000 \times (49.4 + 30.2) \text{ ms}$$

or 13.3 minutes. This is much slower than the time for 10,000 fetches using the hash table, which had buckets smaller than the leaf size here and which needed only one disk access, not two.

However, the time for the ten range searches is

$$10 \times 0.03 \times 40,000 \times (24.3 + 30.2) \text{ ms}$$

or 11 minutes. This is much faster than using sequential reading through the whole file or using a secondary B⁺-tree.

We have just completed a typical estimate made by a file designer. We knew that hashing would be the fastest possible method for individual-record searches. We knew that hashing could not be

used for range searching. We knew that secondary B⁺-trees would not be as good as primary B⁺-trees for range searching.

We might or might not have guessed that secondary B⁺-tree searches would not be as good as sequential reading for 3% of a large file. We probably were unable to guess that the primary B⁺-tree would be better for this particular mix of operations than the combination of hashing for primary searches and sequential reading for range searches. If there had been very many more individual-record fetches, or if the ranges in the range searches had been larger, our answer might have been different.

This should be a convincing argument for the usefulness of estimates. The mix of operations we were considering took less than 25 minutes using the primary B⁺-tree. It took more than 200 minutes for the next best combination of methods. If we had to do something like this every day, we might be able to save a considerable amount of computer time by implementing the method suggested by our estimates.

The Intersection File Once Again

We can use hashing to make the intersection of the Massachusetts hospital file and the BCBS file. We can do this in two ways. First, we assume that there is already a hash-table structure on the BCBS file. Both files have 100,000 400-byte records. There are 70,000 records in common.

Let us suppose that there is virtually no overflow in the BCBS file. (The average chain length, x , is 0.) This would occur with a load factor of 70% and large buckets, for example. Then

$$T_F = r + s + dtt$$

Assuming that $Bkfr = 50$, we obtain $T_F = 32$ ms.

We load the Massachusetts file in memory, 10 megabytes at a time. We then use the records in memory to search for matches using the hash table on the BCBS file. This takes 3200 seconds, or 53 minutes. Solving this problem by sorting both files, and then matching took 2.5 minutes, as we saw in chapter 4. Using a hash table to look up records one at a time is not efficient for this problem. However, a new technique using hashing called *hash partitioning* is efficient.

Hash Partitioning

Let us explain how hash partitioning works on this example. Assume that the hash table on the BCBS file has a load factor, Lf , of 70%. Then the BCBS file takes up 57 megabytes. We divide the hash table into six equal parts, or *partitions*, each a little less than 10 megabytes. This also implies a partitioning of the values of the hash function, which has been calculated from a field such as the social security number of the patient.

We then use the same hash function on the same field on the Massachusetts file. We allocate six areas in memory, corresponding to the six partitions of the values of the hash function. We read in the Massachusetts file and place the records in memory in the area corresponding to the hash-function calculation. When one of the areas fills up with Massachusetts records, we stop reading, and we write the records from that memory area to an area on the disk reserved for the given partition. Then we resume reading the Massachusetts file.

At the end of this process, the Massachusetts file has been partitioned into six parts, each corresponding to one of the partitions of the BCBS file. This has taken us about 28 seconds: 14 seconds for reading in the Massachusetts file and 14 seconds for writing out the records to the six reserved areas on the disk. In this example, the seek times for writing out each piece are not significant.

Then we read in the first partition of the BCBS file to memory. It is already structured as a hash table, with only one subrange of the hash values. One of the partitions of the Massachusetts file corresponds to the same subrange of hash-function values. We then read in the corresponding Massachusetts records, using double buffering to calculate the hash function and compare the records with the BCBS records in the hash table in memory.

If there is any match of a Massachusetts record from the first partition, it will be a match with one of the BCBS records in memory. After the first partition of the BCBS file is matched with the first partition of the Massachusetts file, we write out the records in the intersection. Then we read in the next partition of the BCBS file.

It takes 20 seconds to read in the six partitions of the BCBS file, since there is extra space. It takes 14 seconds to read in the Massachusetts file, since these records have merely been partitioned, not formed into a hash table. It takes 10 seconds to write the intersection file. The total time for this method is 72 seconds. This is the fastest

method so far.

Average Length of Stay in Hospital by Group

In chapter 3 we looked at keeping tables of average length of stay in hospital by medical condition. We assumed that there was a numerical code for each medical condition. This is really a hash table, and the process is the same as the one we used in the matching phase of the hash partitioning: we had a hash table in memory, and we read in records and mapped them to entries in the hash table.

We can therefore say that hashing is used for calculations of averages or sums or standard deviations by group. But the file has no hash-table structure. It is still a pile file.

Section Summary

In summary, if the growth of a file can be predicted, hashing with buckets and chains is an excellent access method. It is fast for fetching, inserting, and deleting, but not for sequential operations. Hashing cannot be used for range queries.

Usually, right after a hash table has been created, fetching takes about one disk access. This is the method of choice for many files and many database management systems. When most of the operations on the file are individual-record fetches, hashing is the optimal file structure.

When there is a mixture of sequential operations, range searches, and individual-record fetches, a B^+ -tree may perform better than a hash table. It is a good idea to make a rough estimate of the time needed for a given mix of operations to decide whether to use hashing or B^+ -trees or some combination.

If the file is likely to grow considerably, hashing with buckets and chains requires frequent reorganization. Nevertheless, most existing file systems follow this method. If you have the opportunity to start from scratch and write your own system, you might try linear hashing, from section 6.2, or bounded index extendible hashing from the appendix to this chapter. These new hashing methods do not need reorganization.

Exercises

6.1.1. Using buckets of size 3 and a hash function $h(x) = x \bmod 5$ and chaining, as in the illustration in this section, enter into an empty hash table the following values, creating chains of buckets as necessary.

42,57,16,52,66,77,12,25,21,33,32,14

6.1.2. Suppose that we start with 100,000 records of 400 bytes each, as in the hospital file. We assume that $Bkfr = 50$ and $Lf = 0.70$, and we set up a hash table. Right after this file is loaded, T_F should be one disk access and the average chain length should be near 0. Suppose that we then run the hospital a few more months and find that we have 400,000 records in the file. The primary area has not changed. What is the new load factor? (It will be greater than 1.) Suppose that the average chain length is now near 3. What will be the new T_F for successful searches?

6.1.3. Suppose that we have one million single-record fetches and ten small-range queries of 3% of the file each on the California file. Which is more efficient, using a hash table for the single-record fetches and sequential reading for the range searches, or using a primary B^+ -tree which has been made optimal for this mix of operations?

6.1.4. Suppose that we have 10,000 single-record fetches and 100 range queries of 0.005 of the California file (one half of one percent). Which is optimal: hashing and sequential reading, hashing and a secondary B^+ -tree, or an optimized primary B^+ -tree?

6.1.5. Suppose that we must make 10,000 single-record fetches and ten small-range queries of 3% of the hospital file (100,000 400-byte records). What is the most efficient file organization: hashing with sequential reading, hashing and a secondary B^+ -tree, or an optimized primary B^+ -tree?

6.2 LINEAR HASHING

Several new methods have been proposed to avoid reorganization of hashed files. One such method is *linear hashing* (Litwin 1981). Linear hashing maintains a constant load factor, even when many new records are added to the file. It does so by incrementally adding new buckets to the primary area. Thus as new records are added, new space for the records in the primary area is also added. The load factor, which is the ratio between the number of records and the number of spaces for records in the primary area, remains constant. Maintaining a constant load factor as the file grows is a revolutionary idea.

Fetching a Record

In linear hashing, the last binary bits in the hash number are used for placing the record. That is, we first calculate a large number from the key, just as before. Then we place the record in a bucket according to the last k binary digits in the hash number. For example, we could begin with eight buckets and use the last three bits in the hash number that was calculated. If the number ended in 000, it would be placed in the first bucket. If it ended in 001, the second; in 010, the third; and so on. This is illustrated in figure 6.3.

When we expand the table, we split an existing bucket which holds all the records which end in a particular k digits into two buckets using the $k + 1$ last digits of the hash number. This way, for example, the bucket with records whose hash number ended in 010 would be split into one bucket with records whose hash number ends in 0010, and another bucket with records whose hash number ends in 1010. After such splits, some of the buckets hold records where k digits are used and some where $k + 1$ digits are used.

The splits are handled following an algorithm which we shall outline in treating insertion. For fetching, we need only know that some of the buckets use k digits, and some use $k + 1$ digits, and we can tell the difference by checking whether or not the last k digits are smaller than a given value, the *boundary value*. The boundary value and the current value of k are kept in the file header. First we shall give a formal algorithm for searching.

000:	8		
001:	17	25	
010:	34	50	
011:			
100:	28		
101:	5		
110:			
111:	55		

34 = 100 010
 17 = 010 001
 5 = 000 101
 28 = 011 001
 25 = 011 001
 50 = 110 010
 55 = 110 111
 8 = 001 000

Figure 6.3. Linear hashing using the last three binary digits of the hash number.

A Search Algorithm for Linear Hashing

1. Calculate the hash function.
2. Look at the last k digits of the hash function. If the last k digits are less than the boundary value, the location is in the bucket labeled with the last $k + 1$ digits. If the last k digits are greater than or equal to the boundary value, the location is in the bucket labeled with the last k digits.
3. Follow overflow chains, as with traditional hashing.

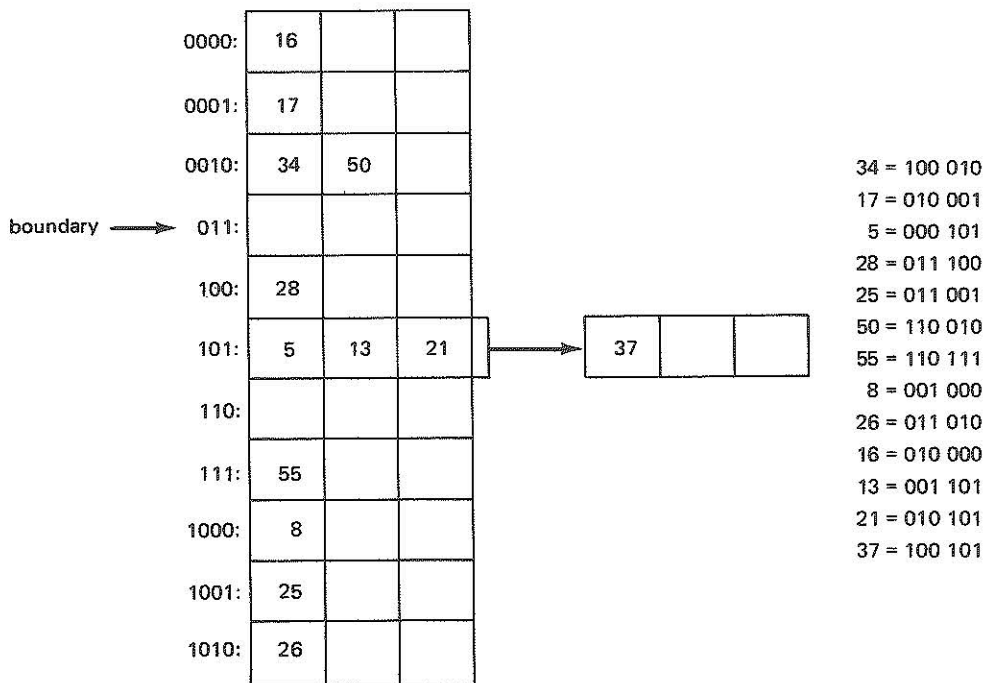


Figure 6.4. Linear hashing using the last three or four binary digits of the hashing number. If the last three digits are less than 011 in binary, use four digits; otherwise use three digits.

For example, if $k = 3$ and the boundary value is 011, as in figure 6.4, and the hash number in binary is 1000 (8 in decimal), we must use the last 4 digits, 1000. On the other hand, if the number is 1101 (13 in decimal), we use only the last 3 digits, 101 (5 in decimal). So the number 1000 is stored in bucket 1000, and the number 1101 is stored in bucket 101.

Since some of the buckets (the ones labeled with k digits) are used for about twice as many records on average as others, the fetch time with linear hashing is not as good as with hashing with buckets and chains for the same load factor and bucket factor. However, it is still pretty close to one disk access.

The time to fetch a record in linear hashing has been experimentally estimated in several cases. For example, if the number of records in a bucket, $Bkfr$, is 50, and the load factor, Lf , is 75%, an aver-

age of 1.05 accesses is needed for a successful search, and 1.27 for an unsuccessful one. That is, under these conditions, we have

$$T_F (\text{successful}) = 1.05 \times (s + r + dtt)$$

$$T_F (\text{unsuccessful}) = 1.27 \times (s + r + dtt)$$

Note that this does not depend on the total number of records. We can enter 100 times the original amount of records estimated to be in the file and get the same performance. This is the advantage that linear hashing has over traditional buckets and chaining. In the traditional method the load factor and, consequently, the fetch time can become very large.

However, the fetch time is a little slower with linear hashing than with traditional hashing *for the same load factor*. This is due to the fact that linear hashing does not evenly distribute the records; the buckets labeled with k digits hold more records on average than the buckets labeled with $k + 1$ digits.

Insertion

We shall illustrate the insertion algorithm for linear hashing by using an initial hash table with eight primary-area buckets with a bucket factor of 10, and a load factor of 70%. We use chaining for overflow for a particular bucket, just as with traditional hashing. When we have 56 records in eight primary-area buckets with $Bkfr = 10$ and possible chains, we have a 70% load factor as desired ($Lf = 0.70$). Now suppose that we add 7 more records to the file, so that we have 63 records. We need another primary-area bucket to keep the load factor near 70%.

In general, we add a new bucket to the primary area after $Lf \times Bkfr$ records are added to the file. So we allocate one more bucket to the primary area after adding 7 records. This is illustrated in figure 6.5. Note that the existing chains are not necessarily removed.

We split the records which were previously in the bucket associated with 000 into two buckets, one for records whose keys hash to numbers ending in 0000 and one for records whose keys hash to numbers ending in 1000. Both categories of records were in the bucket which is now split.

After an additional 7 records is added to the file, we split the bucket associated with 001 into 0001 and 1001. If we imagine the

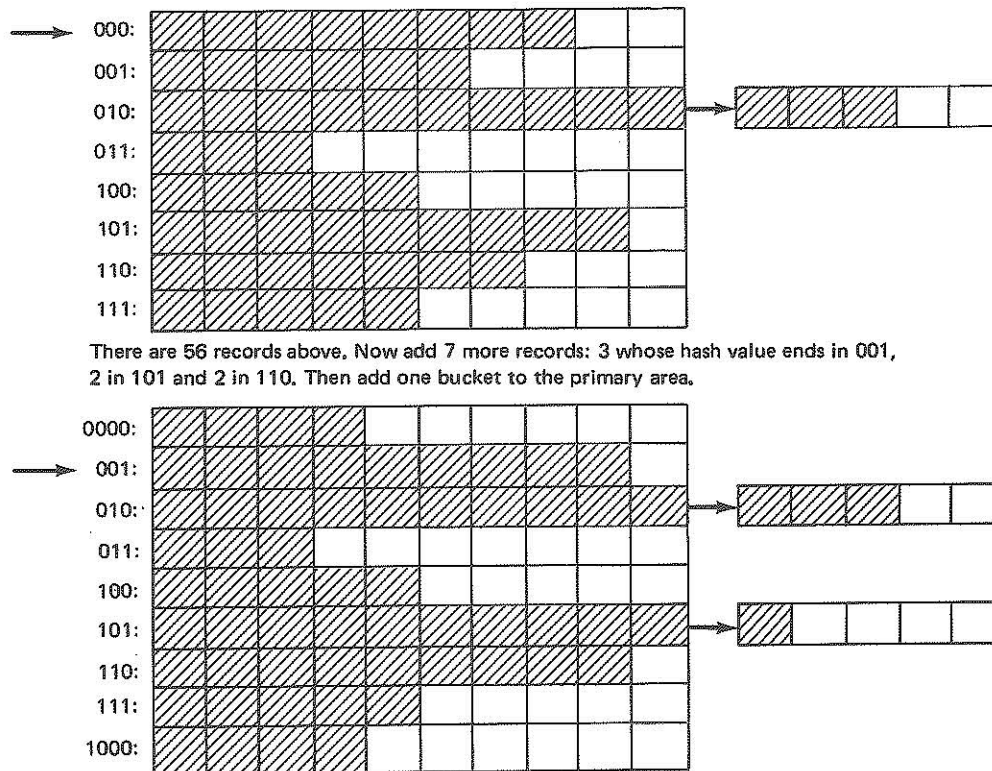


Figure 6.5. Linear hashing table expansion. The load factor is 70%, and the bucket factor is 10. After each seven records is inserted, one more bucket is added to the primary area. Existing chains are not necessarily removed. A marker indicates the boundary between the buckets which use $k + 1$ digits and the buckets which use k digits. The records in the bucket after the marker are split between the old bucket and the new bucket, according to the last $k + 1$ digits in the hash number of their keys.

addresses of the buckets as containing leading 0s as well as the digits we see, note that the addresses of the new and old data buckets remain in order, and the hash number itself provides us with the correct address. We need only one more piece of information: how many of the last binary digits are being used.

There are always either k or $k + 1$ digits used. We need only know the value of k and the first address which uses only k digits (illustrated by a pointer in the diagrams). We have called the first address which

uses only k digits the boundary value. This information can be kept in the file header, as we have suggested.

An Insertion Algorithm for Linear Hashing

1. Search for the correct bucket in which to place the new record.
2. If the bucket is full, allocate a new bucket and insert the record in the new bucket. Add the address of this new overflow bucket to the last bucket in the chain.
3. If there are now $Lf \times Bkfr$ records more than needed for the given load factor, Lf , add one more bucket to the primary area. (This will happen, for example, when exactly $Lf \times Bkfr$ records have been added and none have been deleted since the last primary-area expansion.) Distribute the records from the bucket chain at the boundary value between the original area and the new primary-area buckets. Add 1 to the boundary value.

Experimental results with linear hashing showed that insertion took 2.62 accesses with $Bkfr = 50$ and $Lf = 75\%$. This can be estimated as follows: First we must fetch and modify the correct bucket. With a probability of $1/Bkfr$, we may have to create a new link. (Actually, the load factor will also determine how often a new bucket is to be allocated. With high load factors, overflows are more frequent. We are trying to make the estimate as simple as possible, so we look only at what it takes to make a bucket full.)

In addition, if this is the time to expand the primary area, we must write a new bucket and modify an old one. We may also have to read in an overflow bucket (or in a very unlikely case, several overflow buckets) attached to the old primary-area bucket which is being split.

We get a formula something like this:

$$T_I = T_F (\text{unsuccessful}) + 2r + (1/Bkfr) \times (s + r + dtt)$$

$$+ [1/(Lf \times Bkfr)] \times (s + r + dtt + 2r + s + r + dtt)$$

This is a fetch of the bucket in which to put the new record plus time for adding a chain if this is necessary. In addition, every time the primary area is expanded, we must at least read in the old bucket, change it, and write out a new one. Note that $Lf \times Bkfr$ is the number of records which are added to the file for each index expansion. This assumes that we do much more inserting than deleting, so that $1/(Lf \times Bkfr)$ of the inserts triggers a primary-area expansion. Another way of saying this is that we are not considering the effect of deletions in our estimate.

With $Bkfr = 50$ and $Lf = 0.75$, this is

$$\begin{aligned} T_I &= (1.27 + 1/50 + 2/37) \times (s + r + dtt) + (1 + 1/37) \times 2r \\ &= 74 \text{ ms} = 2.34 \times 31.8 \text{ ms} = 2.34 \times (s + r + dtt) \end{aligned}$$

which is near enough to the experimental results. The point is that it is a bit more than the time for two disk accesses. We shall use Litwin's experimental results for estimating insertion cost, just as we use his results for fetching. Thus,

$$T_I = 2.62 \times (s + r + dtt)$$

Insertion in linear hashing is an inexpensive incremental operation, just as it is in traditional hashing. However, in linear hashing, reorganization is also incremental, and we have counted the cost of expanding the primary area in the insertion cost. That is, when we looked at insertion in traditional bucket and chain hashing, we did not count the amortization of the reorganization cost with the insertion cost. With linear hashing, we have added the amortization of the cost of expanding the table. So the cost of insertion in linear hashing appears larger than in traditional hashing, but it is not.

An Insertion Example

We give an extended example of insertion in linear hashing in order to clarify the algorithm. This example is illustrated in figure 6.6.

Suppose that the bucket factor is 3 and the load factor is 67% (or $2/3$). We begin with four buckets. When eight records have been inserted, the load factor is exactly right. After the eight records are inserted, we repeat the following two steps:

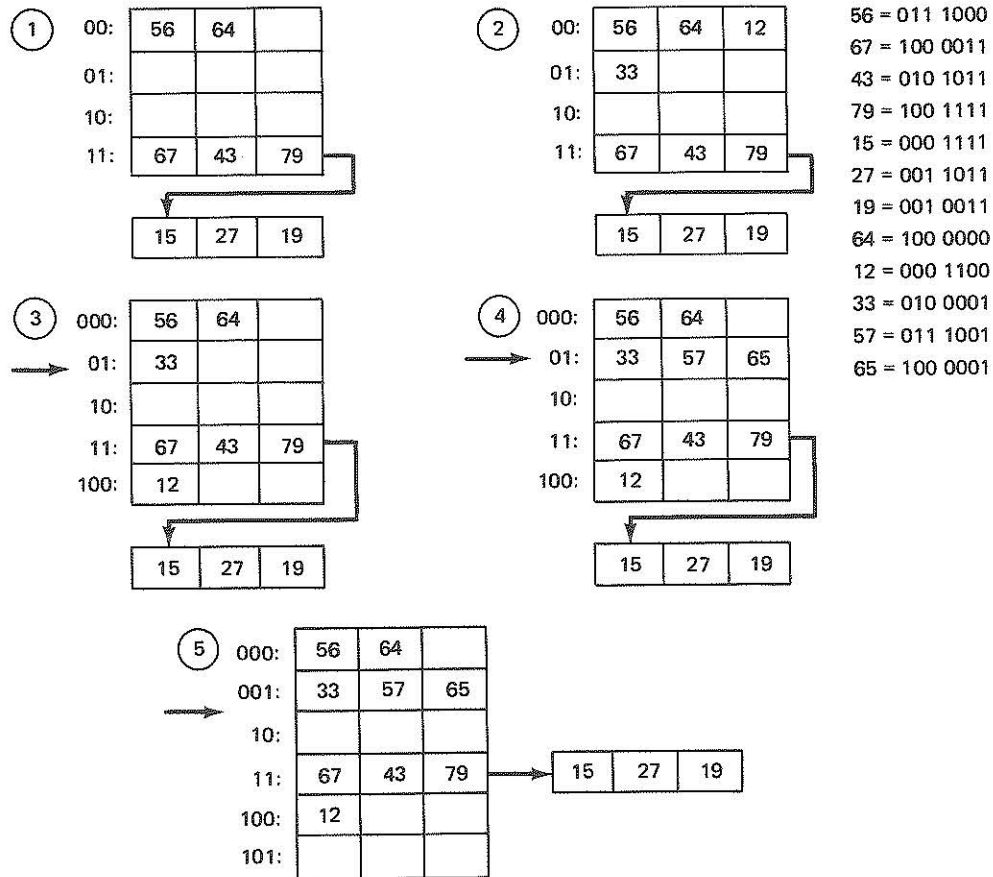


Figure 6.6. A sample problem in linear hashing. We begin with eight records in four primary-area buckets. The load factor is 67% (or 2/3), and the bucket factor is 3. After the initial loading, whenever we insert two records, we expand the table by one bucket.

1. Insert two more records.
2. Add one more bucket to the primary area.

We use the numbers 56, 67, 43, 79, 15, 27, 19, 64, 12, 33, 57, and 65. These numbers were chosen to illustrate the fact that the overflow and the table expansion are independent events; overflow does not trigger table expansion, and table expansion does not necessarily cut down on overflow.

Deletion

We propose the following for deletion. We read in a chain of records. We replace the deleted record with the last record in the chain. If the last overflow bucket in the chain becomes empty, we deallocate it. When the number of records in the file is $Bkfr \times Lf$ less than the number needed for the given load factor, we contract the primary area by one bucket.

Under these assumptions, our time formula for deletion is roughly

$$T_D = T_F \text{ (unsuccessful)} + 2r$$

This is the time to read in the whole chain and modify the bucket containing the record to be deleted as well as the bucket containing the last record. Occasionally, the last bucket in the chain is deallocated, or the table is contracted. This happens so rarely, since there are many more insertions than deletions, that we do not count it at all. We do not even bother to account for the fact that two buckets may be modified.

The algorithm for compressing the table is the exact opposite of the algorithm for expanding it. We keep track of the total number of records in the file, and the total number of primary-area buckets. When we have $Bkfr \times Lf$ fewer records than are needed for the given load factor, we may consolidate the last bucket in the table with the bucket which shares the same last k digits. This is illustrated in figure 6.7.

Thus, linear hashing expands and contracts the primary (non-chained) area gracefully, one bucket at a time, as the number of records increases or decreases. The load factor of the whole table—the number of records in the file divided by the capacity of the primary area of the table—remains nearly constant. Massive reorganizations are never needed.

Allocating Space for Expanding the Primary Area

There is a hidden complication in linear hashing. We normally do not have vast quantities of contiguous space on the disk for allocating to the primary area as it expands. This is also a problem for the pile file, and for the sorted sequential file. The B^+ -tree must allocate more room for new leaves when leaves split.

The file header for all these files will contain a disk allocation directory, which tells where all the pieces of the file are on the disk.

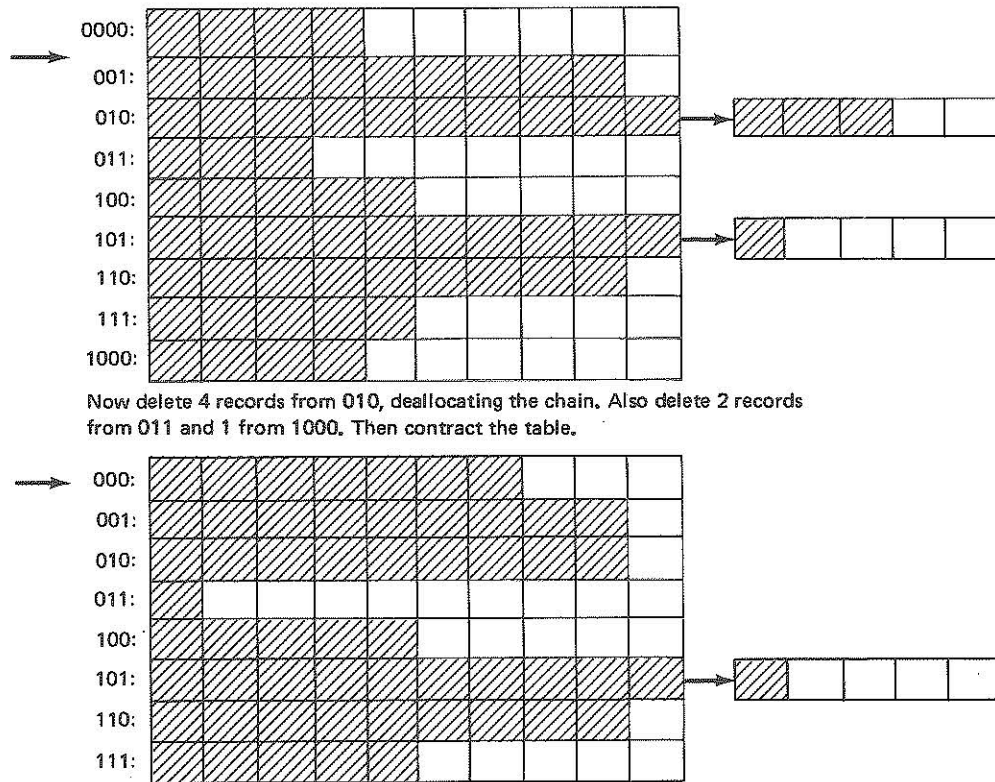


Figure 6.7. Linear hashing table contraction. To keep the load factor constant, the hash table can be contracted by one bucket every time there are $Lf \times Bkfr$ records less than are needed for the fixed load factor desired. In this example, where the load factor is 70% and the bucket factor is 10, we remove one bucket from the primary area whenever we have seven records less than needed for a 70% load factor.

We have called these areas on the disk “extents.” We always assume that large extents are allocated to files as they grow. We assume that a small table indicating the addresses and the sizes of these extents is kept in the file header. Addresses of buckets for linear hashing are calculated as offsets from extent addresses. This is a very minor inconvenience, as the number of large extents for any given file should be small.

Sequential Reading

Hash functions *never* preserve order, so linear hashing cannot be used efficiently for listing a file in order. If we have a sorted list of the values for the key, the time to read the file in order of that key would be

$$T_X \text{ (ordered by list of key values)} = n \times 1.05 \times (s + r + dtt)$$

It does not make sense to try to use the linear-hashing structure without first using a hash function which distributes the data more uniformly. Using the actual data values will not work. The distribution of the values must be nearly uniform for any hashing scheme to have decent space utilization and small numbers of overflow buckets. Real data is never uniform, in either its lower-order bits or its upper-order bits.

As in traditional hashing, it does not pay to use the hash table for sequential reading or for range searches. If order does not matter, we simply read from beginning to end, ignoring the hash structure. If order does matter, we sort the qualifying records after picking them out first.

For an operation such as finding the average salary of all employees, the hash table is less efficient than the pile file, because there is wasted space. For range searches, the B⁺-tree is usually the method of choice. These statements were true for hashing with buckets and chains, and they are also true for linear hashing.

Section Summary

Linear hashing does not have to be reorganized. The primary area grows and shrinks one bucket at a time. It keeps a nearly constant load factor, determined by the person setting up the file parameters. Thus, it is space efficient and time efficient. Like traditional hashing, linear hashing is not good for sequential processing or range queries.

Linear hashing, like traditional hashing, is excellent for single-record fetches. It is also efficient for insertion and deletion of records.

Linear hashing is so new that not very many systems are now using it. However, it is generally agreed that this method, and others similar to it, (like the method in the appendix to this chapter) are superior to the more traditional methods which have to be reorganized frequently.

Exercises

6.2.1. Enter these records using linear hashing with a bucket factor of 2 and a load factor of 50% (or $1/2$). Start with four buckets (last two binary digits) and four records. When the fifth record is loaded, add another bucket. Add one bucket for each record insertion after that. First convert the keys to seven-digit binary numbers. Draw the buckets. Note that there is no overflow. This is likely with a load factor of 50%. However, half the space is wasted.

42, 57, 16, 52, 66, 77, 12, 25, 21, 33.

6.2.2. Using the same bucket factor and load factor as in the previous problem, make up a list of ten numbers which will cause one chain which has two overflow buckets sometime in the insertion process. Draw the hash table at this point.

6.2.3. As in the example in figure 6.6, load the first eight records into an initial hash table of four buckets with $Bkfr = 3$ and $Lf = 67\%$. After that, repeat inserting two records, and then expanding the table by one bucket. Draw the linear-hashing table right before each expansion. Use the following hash values: 42, 57, 16, 52, 66, 77, 12, 25, 21, 33, 4, 5, 6, 7. (The first ten values are the same as in problem 6.2.1.)

6.3 CONSTRUCTING A HASH TABLE

Since traditional hashing involves frequent reorganization, it is useful to be able to construct a hash table from an existing file very quickly. This is an operation which may have to be done every day or every week with a rapidly growing file.

If we wish to construct a hash table for an already existing file and we have a hash function which allows us to use only one disk access to fetch the correct bucket, the construction time will be at least

$$n \times (r + s + dtt + 2r)$$

This is a one-access fetch and a modify for each record. We assume that we have an empty table to start with, so that overflow is a remote possibility.

In the case of the California income tax file, where $n = 6,000,000$ and we use 2400-byte buckets, the time for constructing the file is at least

$$6,000,000 \times (41.7) = 2.5 \times 10^8 \text{ ms} = 250,200 \text{ sec}$$

which is about 69.5 hours, or nearly 3 days. This does not count reading in the file to be hashed.

So let us look for another method of loading a file into a hash table.

We suggest two methods. One method is to read in the file, attach the hash value, and sort by hash value. If we are using traditional hashing with chaining, the last pass of the sort can be used to put all records with the same hash value in a bucket. The second method involves partitioning the space of possible hash values, instead of sorting.

Let us examine this first process, sorting by hash value, for both traditional hashing and linear hashing for the California income tax file.

Space for the File

First, in all strategies we must estimate the amount of space the hashed file will use. The number of buckets in the primary area, M , for traditional or for linear hashing is

$$M = n / (Bkfr \times Lf)$$

This will have to be calculated several times in the life of a traditional file, or every time it is reorganized. For linear hashing, this is done once, at the time it is decided to take an existing file with some other organization and convert it to a linear-hashing file.

Buckets and Chaining

Suppose that we have decided to have buckets with a capacity of 50 records, as before, and that we have Lf , the load factor, equal to 70%. Then we expect $6,000,000 / (50 \times 0.7) = 171,429$ buckets and we arrange a hashing function with values between 0 and 171,428. We read in the records, with one disk drive, sorting by hash function into 240 segments, as in chapter 4 (or a little more to allow for the 18 bits for the hash value which is added to each record).

We then make two more passes to do a merge sort. This took an hour and a half with one disk drive. We need a little extra time in the last pass because we shall be leaving space in most of the buckets and constructing some chains to an overflow area. Still, it will be much shorter than the 3 days it takes for successive insertions.

Linear Hashing

Essentially the same process is used for linear hashing. The number of digits for a given load factor must be calculated ahead of time. If we use 18 digits ($2^{18} = 262,144$), we shall have extra space and a lower initial load factor. In this case with $Bkfr = 50$ and 262,144 buckets, we would have enough room for 13 million records. We probably do not want to allocate so much space for our hash table. If we use 17 digits we shall have a higher load factor and too many chains.

So we have to find out where to switch between 17 and 18 digits. We develop an algorithm to do this in general.

Let us arrive at a method for allocating space for linear hashing when we want exactly M buckets and

$$2^k \leq M < 2^{k+1}$$

or

$$k \leq \log M < k + 1$$

In this case we know that we shall be using k digits for some of the records and $k + 1$ digits for the others.

Look at the table in figure 6.8. The total number of buckets is $M = 11$. The first $2^k = 2^3 = 8$ buckets include the first buckets using $k + 1 = 4$ digits and all the buckets using $k = 3$ digits. The number of buckets at the end of the table using $k + 1$ digits is $M - 2^k$.

Let y be the number of buckets using $k + 1$ digits. Then $y = 2 \times (M - 2^k)$. There will be $M - 2^k$ such buckets at the beginning of the table and $M - 2^k$ at the end of the table. The number of buckets using only k digits is $2^k - (M - 2^k)$ or $2^{k+1} - M$.

0000:	16		
0001:	17		
0010:	34	50	
boundary: → 011:			
100:	28		
101:	5	13	21
110:			
111:	55		
1000:	8		
1001:	25		
1010:	26		

$2^k = 8$
 $M - 2^k = 3$

Figure 6.8. Here we have 11 primary-area buckets for linear hashing. The number of buckets at the end of the table using four digits instead of three digits is $11 - 8$, or in general, $M - 2^k$ digits.

The Number of Digits in a Linear-Hashing Table

1. First find M , the number of buckets in the primary area.

$$M = n / (Bkfr \times Lf)$$

2. We define “ k ” by $k = \text{floor}(\log M)$.
3. For hash numbers whose last k digits are less than $M - 2^k$, use the last $k + 1$ digits for the sort criterion.
4. For hash numbers whose last k digits are greater than or equal to $M - 2^k$, use the last k digits as the sort criterion.

We can figure out the place where we switch between 18 and 17 digits in our example, using this algorithm. We make exactly 171,429 buckets, as before. Since $2^{17} = 131,072$, and $171,429 - 131,072 = 40,357$, we have $2 \times 40,357 = 80,714$ buckets for hash numbers with 18 digits. That is, we begin with 40,357 buckets with 18-digit hash numbers starting with 0. Then we have 90,715 buckets with 17-digit hash numbers and 40,357 buckets with 18-digit hash numbers starting with 1.

When sorting, we simply calculate the hash number, look at the last 17 binary digits, and if the number is greater than or equal to 40,357, attach a leading 0. Otherwise the number will be whatever the last 18 digits is.

Partitioning

An alternative to sorting is partitioning. We showed how to use partitioning to create intersection files in section 6.1. Here we show how to apply this new method to creating a hash table for an existing file.

We make a partitioning of the hash function values into some number, say nsg intervals, where we have estimated that $1/nsg$ of the file will fit in memory, with a given load factor. We use nsg , since this is the number of segments, each approximately the size of memory, into which we partition the file. We also take into account the load factor, assuming only Lf of the space in each segment is filled with records.

Then we read in the file, one segment of memory size at a time. We fill memory. Within memory, we partition the segment into nsg pieces, corresponding to the intervals we have decided upon. We then write out each of the nsg pieces to nsg different places on the disk. We have allocated space on the disk for each interval of values. This will require nsg disk accesses for each segment we process.

When the records have thus been partitioned, we read in each collection of records for a given interval separately. We use these collections to construct that part of the hash table in memory. We then write out this part of the hash table to disk.

With one disk drive this takes two reads of the entire file, two writes of the file, and nsg^2 accesses during the first write and nsg accesses for each of the reads and for the second write. That is, the total number of disk accesses is $nsg^2 + 3 \times nsg$.

This method is very close in time to sorting and merging using one disk drive. With the sorting, we created sorted segments and then merged them, using at least two reads and writes of the whole file. With sorting, the large number of disk accesses comes during reading in for merging.

Section Summary

It helps to sort by hash value or else to partition by hash value, in order to construct a hash table from existing large files. If we try to construct the table by successive insertions, it will take too much time.

In traditional hashing, reorganizing occurs often. So it is very important to have an efficient way to create new hash tables.

For linear hashing, we have to create the table only once. Still, we can save a considerable amount of time by sorting by the bits expected to be used in the table, instead of doing successive insertions. We gave an algorithm for deciding how many bits to use for creating a linear-hashing table.

Exercises

6.3.1. Suppose that we have $Lf = 67\%$ and $Bkfr = 3$ and we have 24 records whose hash values are below. What numbers do we use to place them for linear hashing?

42, 57, 16, 52, 12, 25, 21, 33, 32, 14, 50, 49, 47, 7, 56, 13, 62, 27, 31, 41, 30, 4, 10, 6.

APPENDIX: BOUNDED INDEX EXTENDIBLE HASHING

In addition to linear hashing, another new method of hashing which does not require frequent reorganization is *bounded index extendible hashing* (Lomet 1983). This is a refinement of *extendible hashing* (Fagin et al 1979).

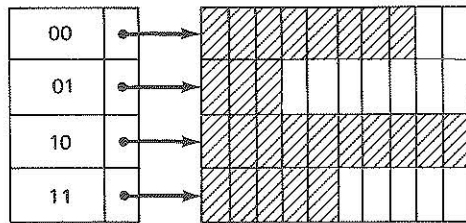
The advantage of bounded index extendible hashing over linear hashing is that there are no overflow chains. Linear hashing has overflow chains which are concentrated in those buckets labeled with the smaller number of digits (k instead of $k + 1$), so the average number of disk accesses is usually slightly higher than one. We begin by describing plain extendible hashing.

Extendible Hashing

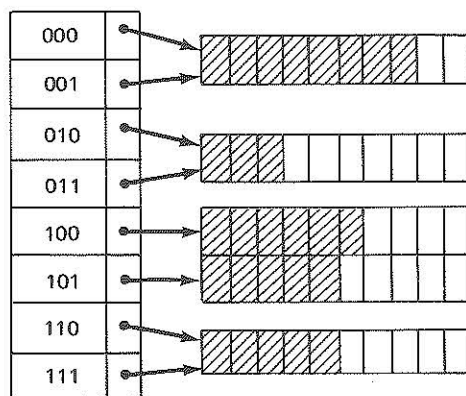
Extendible hashing works like this. First calculate the hash function, as usual. We shall be using the *first* binary digits instead of the *last* binary digits. We shall have buckets as before, but we shall not use any chaining. Instead, we have an index to the hash table. Each index entry contains the address of a data bucket. The number of entries is always a power of 2. If the number of entries is 2^j , the first j binary digits of the hash number are used to find the correct index entry. We call j the *level of the index*.

Several index entries may contain the same data-bucket address. This means that the hash numbers of the keys of the records in that bucket agree only on i digits ($i < j$), or equivalently, all the records with those same first i digits in the hash number fit into one data bucket. Thus, the number of index entries is usually larger than the number of data buckets. We call i the *level of the data bucket*. If there is only one index entry for a given data bucket, $i = j$.

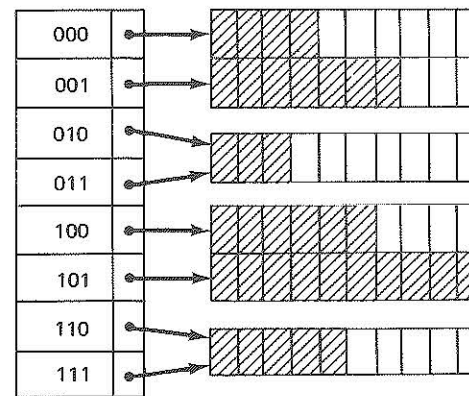
There are two cases to consider when a data bucket overflows. First, if $i = j$, the index must double. Usually we can then distribute the records into two buckets according to the first $i + 1$ digits. If all the hash numbers of the records have the same first $i + 1$ digits, we must double the index again, and so on. Hash functions make distributions of numbers nearly uniform so that long sequences of index doublings should not occur.



(a) Overflow in bucket 10
table doubles; add one bucket.



(b) Add three records to the
first bucket; overflow does
not expand table.



(c) Structure after
second overflow.

Figure 6.9. Extendible hashing. When a bucket overflows, the index must sometimes be doubled. Sometimes, as in the second overflow, it is necessary only to allocate a new bucket and change an index entry.

Second, if $i < j$, again usually we can distribute the records into two buckets according to the first $i + 1$ digits. In this case the index does not double. Half of the index entries which previously contained the address of the old bucket will be changed to contain the address of the new bucket. If, however, the records have hash values with the same first $i + 1$ binary digits and $i + 1 < j$, we must create an empty bucket (for the nonused $i + 1$ digits with the same first i digits) and change the index entries and then look at the first $i + 2$ digits, and so on. If the hash numbers agree on j digits, we must double the index.

Extendible hashing illustrated in figure 6.9. We have four buckets with $Bkfr = 10$. They are labeled with the four combinations of two binary digits: 00, 01, 10, and 11. When one of these buckets overflows,

the index is doubled. Now the bucket holding records whose key hashed to 00 is referred to by two index entries, 001 and 000. The bucket with overflow, 10, has been split into 100 and 101, with one index entry for each of these buckets. There is no chaining.

When the bucket holding records whose hash value begins in 00 overflows, we need only allocate a new bucket and change one index entry, as in the last drawing in figure 6.9. The records whose hash value begins in 000 are kept in the old bucket, and the new bucket holds records whose hash value begins in 001. We assume that not all these hash numbers had the same first three digits. In this case only the index entry for records whose hash value begins with 001 is changed.

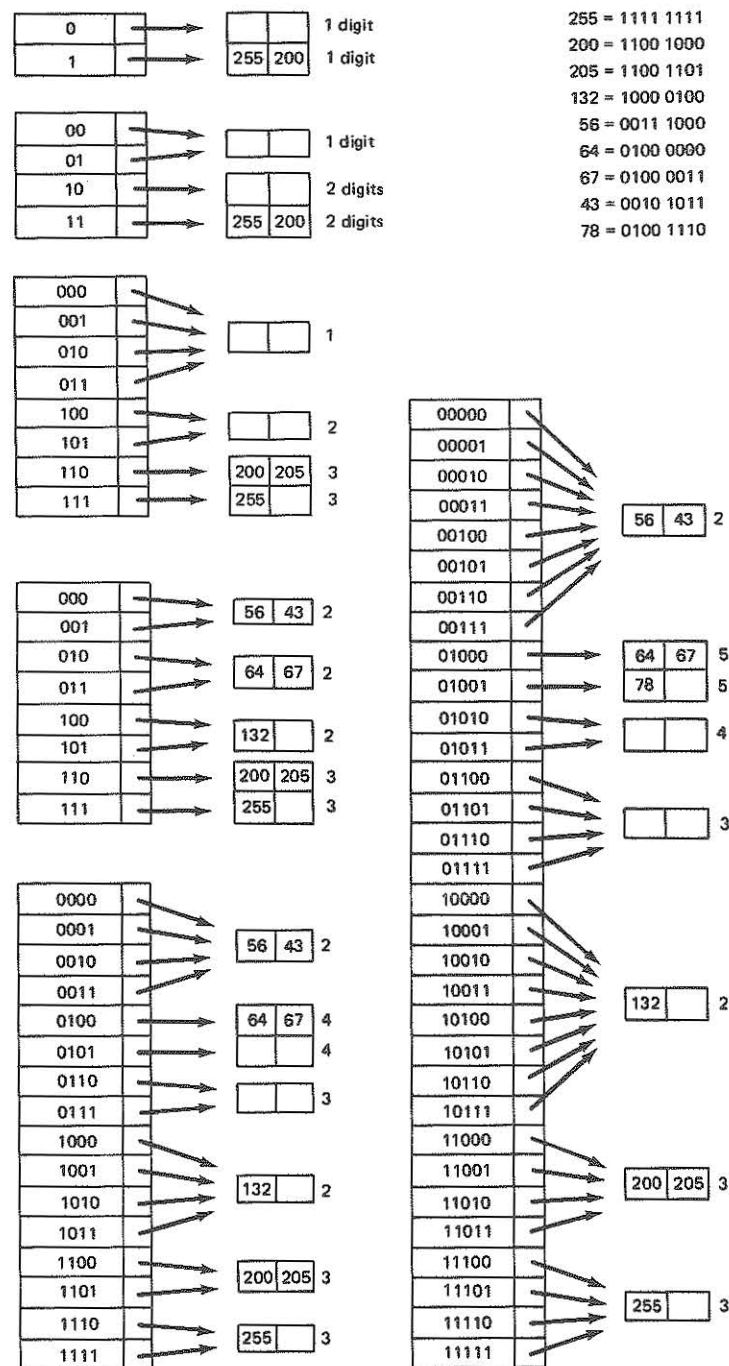
Here, the index takes up space. With linear hashing and traditional bucket-chain hashing, there is no index. This can be a problem in extendible hashing. Let us estimate how much space the index takes up.

Suppose that we have the California income tax file of 6 million records with 400 bytes each. Suppose that we want a 70% load factor and a bucket factor of 50. This means that the number of buckets will be

$$6,000,000 / (50 \times 0.7) = 171,429$$

This is between 2^{17} and 2^{18} buckets. Each directory entry contains a 4-byte address. The size of extendible-hashing tables is always a power of 2. Let us assume we have a table of size 2^{18} . Thus, the index takes $2^{18} \times 4$ bytes or a megabyte of storage. This is not bad. It gives us a one-disk-access method as long as we have at least one megabyte of memory available. But the size of the extendible-hashing table depends heavily on the uniformity of the distribution of hash values. If only one bucket overflows, we may need to double the table. There is some danger in extendible hashing that there will be several extra factors of 2 in the table size because of uneven clustering of hash values.

We illustrate this pathological case in figure 6.10. Here we use eight-digit binary numbers. We attempt to successively insert the numbers 255, 200, 205, 132, 56, 64, 67, 43, and 78. The numbers 64, 67, and 78 start with the same first four digits. But we have a bucket factor of 2 in this example. Thus, we need five digits, or a table with 32 entries, to place these nine records. We must also create several empty data buckets.



If the index uses too much space, it cannot fit in memory, and extendible hashing becomes a 2-disk-access method like the B^+ -tree, but as with all hashing methods, it cannot be used for range searches. Thus, there would never be any reason to choose extendible hashing over B^+ -trees for large files.

There are two unsettling aspects about extendible hashing. One is this possibility of poor space utilization in the index. The other is the nonincremental changes which must be made in a large table, possibly kept on disk, when the table doubles. Both of these problems are solved by Lomet's variation, called bounded index extendible hashing.

Bounded Index Extendible Hashing

In bounded index extendible hashing, a buffer is chosen for the index of the file, which is then fixed in size. Even for quite large files, 16 or 32 blocks should be adequate. The index is only allowed to expand enough to fill the buffer, using the process in the original extendible hashing method. After this, the file grows by doubling the data-bucket size, so that the number of blocks in a data bucket is a power of 2.

We give an illustration in figure 6.11. Here we assume that there are only four blocks in the index. They are labeled 00, 01, 10 and 11. The first two digits of the hash number determine the block of the index. Then within that block are pointers to data buckets labeled by the next three digits in the hash number. The index also records the power of 2 for the number of blocks in a given data bucket. That is, a number, w , is recorded in the index if there are 2^w blocks in the given data bucket. The next w digits are then used to give the offset from the address of the bucket which can be used to calculate the block address for the given record.

Let us use this illustration to show how fetching is done in general. Look at the picture in figure 6.11 while reading this algorithm.

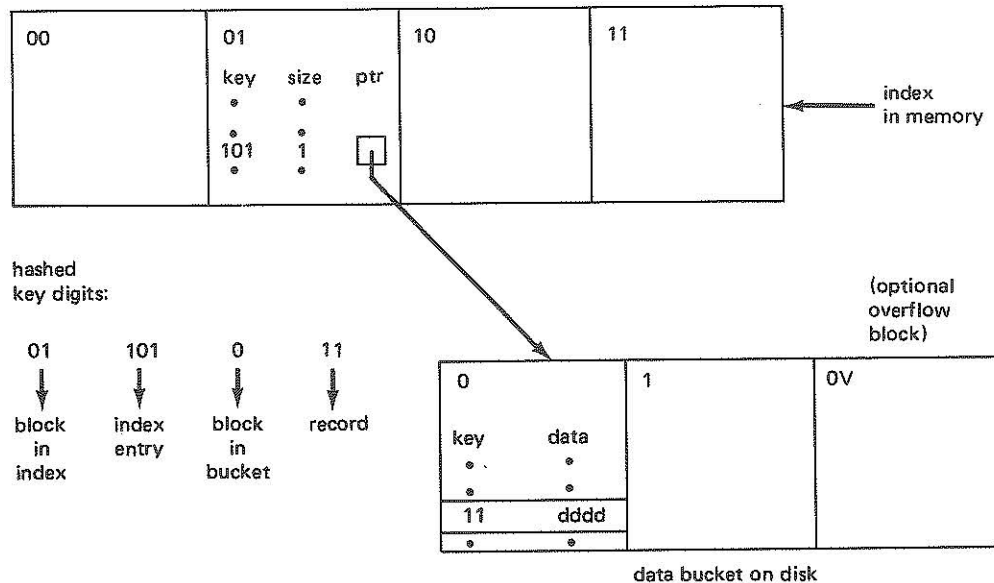


Figure 6.11. Bounded index extendible hashing. The first two digits yield the address of the block in memory containing the correct table entry. The next three digits give the address of the data bucket. The sixth digit locates the correct block within the data bucket. There is also an overflow area stored with each bucket, so that premature splitting does not occur.

A Bounded Index Extendible Hashing Search Algorithm

1. Calculate the hash number.
2. If there are 2^y blocks in the index, use the first y binary digits of the hash number to locate the block with the correct index entry. The entire index is kept in memory while the file is in use.
3. Within the correct block, z more binary digits designate the index entry. Use the next z digits to find the correct index entry. This will yield the address of the correct data bucket. Also, there will be a number, w , indicating the log base 2 of the number of blocks in the data bucket. The number of blocks in the data bucket is always a power of 2.

4. Use the next w digits of the hash number to calculate the offset of the correct block in the correct bucket. We have now used $y + z + w$ of the leading binary digits in the hash number to find the address of the block containing the record we seek.
 5. Make the one and only disk access to obtain the correct record. (If an overflow area is kept in the data bucket and the record is not in the correct block, search the overflow area.)
-

When a block in a bucket becomes too full, the bucket is doubled, and the exponent in the index in memory is incremented by one. When this happens, the bucket contents are copied to contiguous blocks in a larger bucket somewhere else on the disk. In this way, search becomes a one-disk-access method.

If we refine this method by allowing a certain number of overflow blocks for each bucket, stored with the bucket, the search time is sometimes more than one disk access. The reason for making this refinement is to avoid doubling a bucket prematurely. One block in the bucket may have a slight amount of overflow, and the others may be sparse. In this case it might be good to postpone doubling the bucket size.

Appendix Summary

Extendible hashing is a method for gracefully increasing the size of a hash table by occasionally doubling the index. This is a one-disk-access method if the index fits in memory, and a two-disk-access method if it does not. Bounded index extendible hashing is a variation on extendible hashing which makes it about one disk access even for very large files. It is exactly one disk access if no overflow blocks are included in the buckets.

Bounded index extendible hashing is likely to have slightly better fetch times than linear hashing, since there are no overflow chains, and since in linear hashing, the values are not distributed evenly among the buckets, even with a good hash function.

Exercises

6.A.1. Assume that we have set up bounded index extendible hashing with four blocks in the index and four entries in each index block. Suppose that a data block has space for two records. Suppose that the hash function yields eight-digit binary numbers. The first two digits determine the block in the index. The second two determine the address of the bucket.

Enter records with the following hash numbers, doubling the data bucket when necessary. Draw the result.

255, 200, 56, 64, 155, 67, 43, 12, 205, 225, 132, 128, 144, 79

CHAPTER SUMMARY

Hashing is the best file structure for single-record fetches. When a large number of individual records must be fetched, as with automatic teller machines or airline reservations, this is the best method.

Insertions and deletions are also very fast, so hashing is a good method for dynamic files. If the insertions and deletions balance each other out, nothing more is necessary. However, most files tend to grow. There are more insertions than deletions. Thus, traditional hashing structures must be periodically reorganized to retain their excellent performance. Newly proposed hashing schemes do not have this problem.

Linear hashing and bounded index extendible hashing do not need to be reorganized. Both methods gracefully and incrementally change the primary area as the file grows and shrinks.

In linear hashing, as new records are added, new buckets are appended to the primary area. The load factor is kept constant. The last k or $k+1$ binary digits in the hash number yield the address of the correct data bucket in the primary area. There are overflow chains, as with traditional hashing, but since the load factor is constant, they never become so bad as to make the performance unacceptable.

The buckets using k digits have more overflow than the buckets using $k+1$ digits, since each such bucket corresponds to twice as many records. This makes the average fetch time slightly larger than in traditional hashing for a given load factor.

In bounded index extendible hashing, for a while the index doubles as the file grows. After the index reaches a predetermined size, which easily fits in memory, the primary-area buckets double in size when more space is needed. The first few digits of the hash number give the location of the index entry in memory, the next few determine the address of the data bucket, and the subsequent digits yield the offset of the correct block within the correct bucket.

No hashing method is good for sequential operations, or for range searches. For range searches, it is best to use B^+ -trees. For sequential operations, such as finding an average, where order does not matter, the pile file is best. Even using a hashed file and ignoring the hash structure to do averages will be slower because a hashed file, like a B^+ -tree, has wasted space.

For creating ordered lists, such as mailing lists ordered by ZIP code, the sorted sequential file is best. For a mix of operations involving individual-record fetches, range searches, and ordered lists, it is best to try to make a rough estimate using the techniques of this book to decide on a file organization.

For creating a hash table from an already existing large file, as happens initially for the linear-hashing table or repeatedly for the traditional hashing method, we need to sort the records by their hash value and create the table in the last pass. Successive insertion will be too slow. Hash partitioning can be used instead of sorting, but it offers no significant advantage over sorting.

Formulas

We use M for the number of buckets in the primary area. We use $Bkfr$ for the number of records a bucket in the primary area can hold. The load factor, Lf , is the number of records in the file divided by the number of spaces for records in the primary area. That is,

$$Lf = n / (M \times Bkfr)$$

For hashing with buckets and chains (before significant overflow), single-record fetches are about one disk access. That is

$$T_F = s + r + dtt$$

If there has been a lot of overflow, so that the average number of chains, x , is significant, we use

$$T_F \text{ (successful)} = (1 + x/2) \times (r + s + dtt)$$

and

$$T_F (\text{unsuccessful}) = (1 + x) \times (r + s + dtt)$$

We assume that we have chains of overflow buckets, each the same size as a primary bucket.

For insertion and for deletion, we use

$$T_I = T_D = T_F (\text{unsuccessful}) + 2r$$

We have taken here the simplest formula. This assumes that for deletion, we replace the deleted record with the last record in a chain. So the whole chain is read in, and one or two buckets must be modified. We assume that there is usually just one bucket to modify.

For insertion, we simply assume that we make an unsuccessful search and insert the record in the correct bucket. Actually, we may need to add a new overflow bucket.

For linear hashing with $Bkfr = 50$ and $Lf = 0.75$, we have experimental results

$$T_F (\text{successful}) = 1.05 \times (r + s + dtt)$$

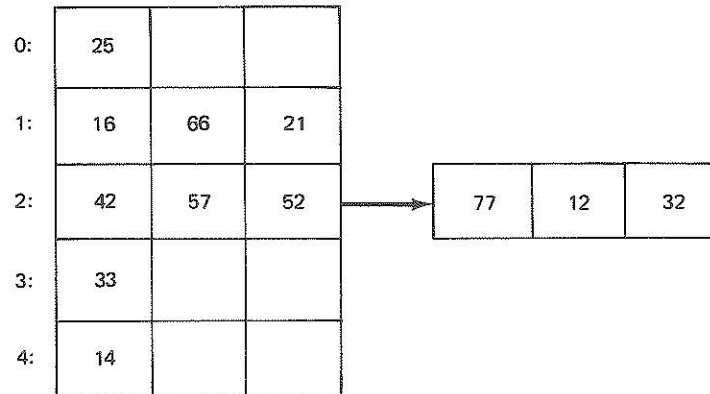
$$T_F (\text{unsuccessful}) = 1.27 \times (r + s + dtt)$$

and

$$T_I = 2.62 \times (r + s + dtt)$$

ANSWERS

6.1.1. Hashing table with $Bkfr = 3$ and $h(x) = x \bmod 5$.



6.1.2. We know that

$$0.70 = Lf = n / (M \times Bkfr)$$

The other variables do not change, but n becomes 4 times as large. Thus, the new Lf is $4 \times 0.70 = 2.8$.

If x , the average chain length, is 3,

$$T_F = s + r + dtt + (3/2) \times (s + r + dtt)$$

At this point, hashing has become less efficient than a B^+ -tree access. Since hashing is usually chosen because a B^+ -tree access is too slow for some mix of operations, it is now time to reorganize and read the whole file into another larger hashing table in order to restore the smaller load factor and reduce the average chain length.

6.1.3. One million single-record fetches using the hashing table takes 530 minutes, or 100 times the amount of time we needed for 10,000 fetches. The sequential read time for the ten range queries is still 200 minutes.

To find the optimal primary B^+ -tree, we solve

$$0 = \frac{1,000,000}{2857} + \frac{25.5 \times 10^9}{x^2}$$

and obtain $x = 9600$ bytes, or about 4 blocks. Since we have the same number of range queries and a larger number of single-record fetches, we expect to have smaller buckets. In this case $dt = 3.4$ ms and $bk = 360,000$ buckets. It takes 880 minutes to make 1 million single-record fetches and 50 minutes for the range queries.

It is faster to use a hash-table structure for single-record fetches and sequential reading for the range queries than to use an optimal primary B^+ -tree for this mix of operations.

6.1.4. For 10,000 single-record fetches we need 5.3 minutes using the hash table, as before. For 100 range queries using sequential reading, we need 100×20 minutes, or 2000 minutes.

For a secondary B^+ -tree, the time is

$$100 \times 0.005 \times 6,000,000 \times 25.1 \text{ ms}$$

or 1255 minutes. The B^+ -tree is more efficient than sequential reading for 0.005 of this file.

To find the optimal primary B^+ -tree, we use

$$10,000/2857 - 0.5 \times 3.5 \times 10^9 \times 24.3/x^2 = 0$$

We obtain $x = 110,400$ bytes, approximately, or around 46 blocks. Here $dt = 38.6$ ms, and $bk = 31,700$ buckets. So the time to make individual-record fetches is 880 seconds, or 14.7 minutes, slower than hashing. The time for the range searches is $0.5 \times 31,700 \times (24.3 + 38.6)$, or 997 seconds, or 16.6 minutes, smaller than the secondary B^+ -tree by a considerable margin.

6.1.5. The time for using the hash file with the same size buckets is the same no matter how large the file is. So finding 10,000 single records takes 5.3 minutes. But doing the sequential reads is faster with a small file. Ten sequential reads of the hospital file takes 140 seconds, or 2.3 minutes.

Using a secondary B^+ -tree for range searches is also faster, since 3% of the file is a smaller number of records. The time for the ten range searches is

$$10 \times 0.03 \times 100,000 \times 25.1 \text{ ms}$$

or 753 seconds, or 12.5 minutes. It is faster to do sequential reading.

The optimal primary B⁺-tree is still calculated as the optimal value of

$$10,000 \times (49.4 + dtt) + 0.3 \times bk \times (24.3 + dtt)$$

Here

$$bk = \frac{100,000}{(x/400) \times \ln 2} = \frac{5.77 \times 10^7}{x}$$

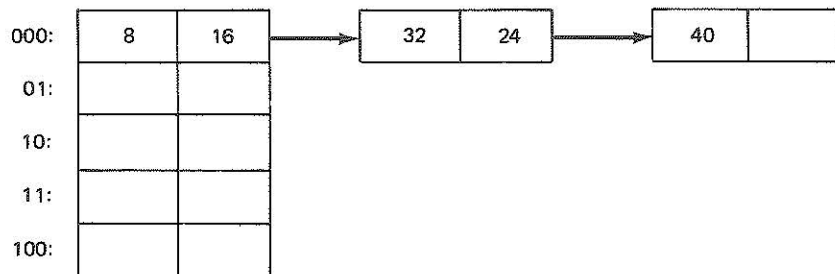
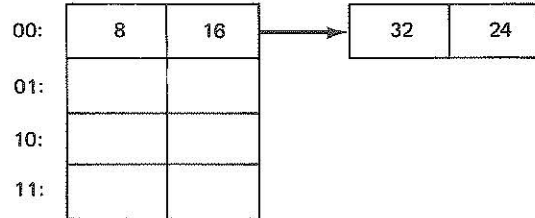
So we have

$$3.5 - 0.3 \times 5.77 \times 10^7 \times 24.3/x^2 = 0$$

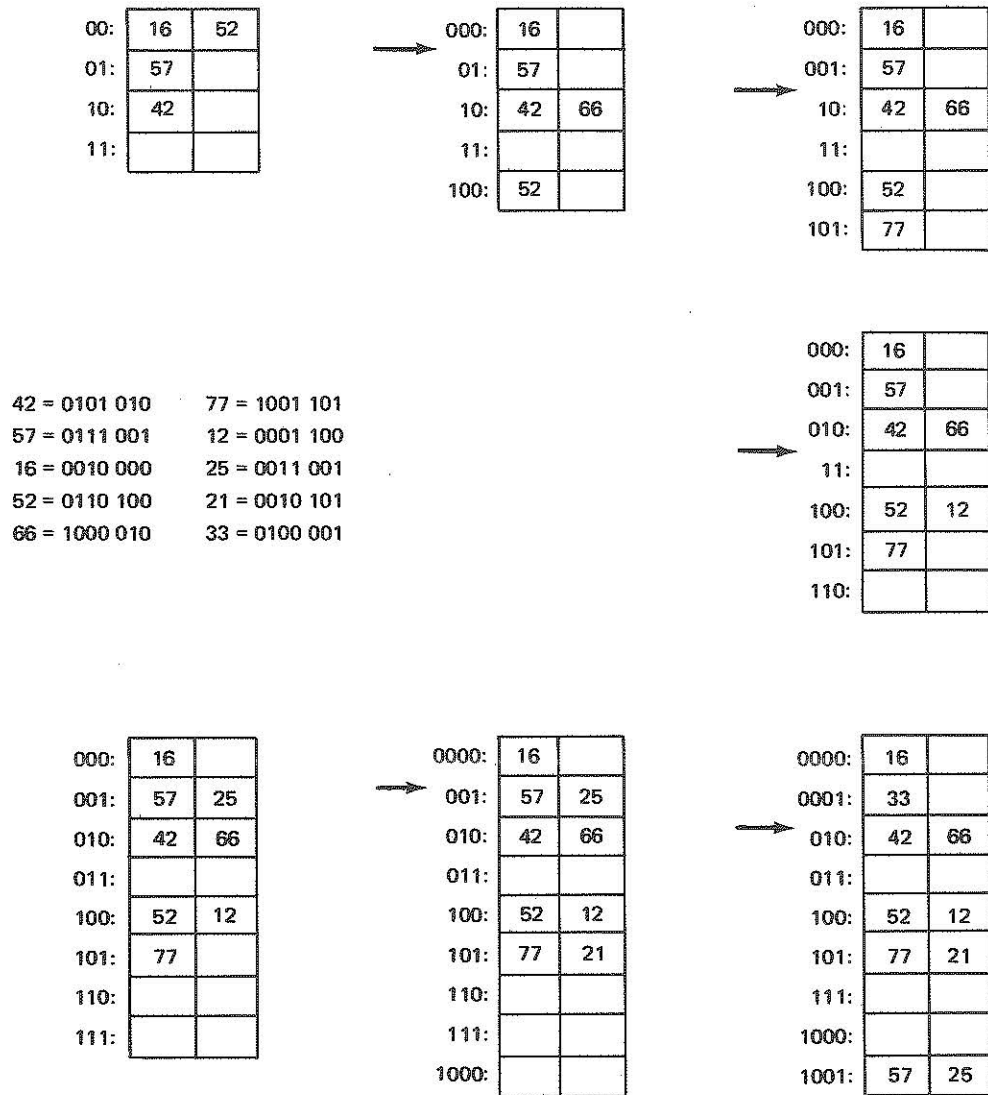
or x is about 14,400 bytes, or 6 blocks. In this case the time for the 10,000 fetches is 9 minutes, and the time for the range queries is about 35 seconds. So the optimal primary B⁺-tree is best for this problem.

6.2.1. On next page (after 6.2.2.).

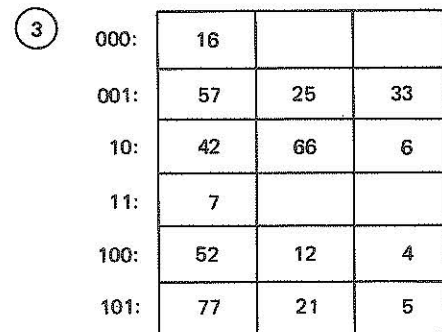
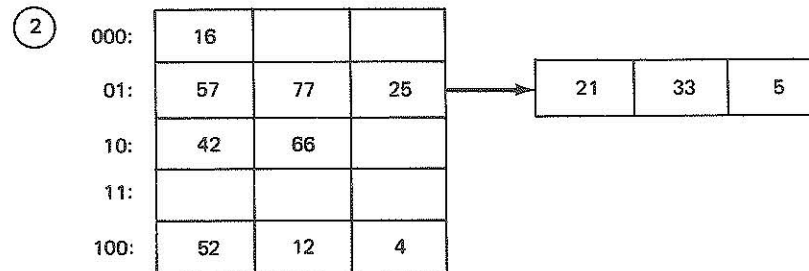
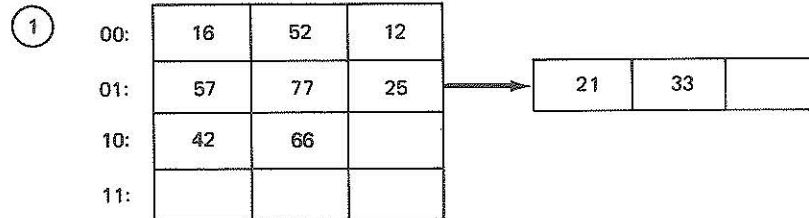
6.2.2. Just make up an example where several hash values share the same low-order digits, such as the following: 8, 16, 32, 24, 40, and 48.



6.2.1. Linear hashing with a bucket factor of 2 and a load factor of 50%.



6.2.3. Linear hashing with a bucket factor of 3 and a load factor of 67% ($2/3$).



6.3.1. Constructing a linear-hashing table from an already existing file where we have 24 records. The bucket factor is 3 and the load factor is 67%.

$$b = \frac{24}{3 \times 2/3} = 12$$

$$x = 3$$

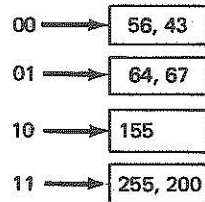
$$b - 2^x = 4$$

If the last three digits are less than 4 (i.e., 000, 001, 010, or 011), use the last four digits.

number	binary	use	number	binary	use
42	10 1010	1010	47	10 1111	111
57	11 1001	1001	7	111	111
16	1 0000	0000	56	11 1000	1000
52	11 0100	100	13	1101	101
12	1100	100	62	11 1110	110
25	1 1001	1001	27	1 1011	1011
21	1 0101	101	31	1 1111	111
33	10 0001	0001	41	10 1001	1001
32	10 0000	0000	30	1 1110	110
14	1110	110	4	100	100
50	11 0010	0010	10	1010	1010
49	11 0001	0001	6	110	110

6.A.1. Bounded index extendible hashing.

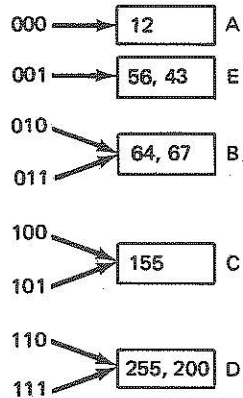
1. To fill the index, we begin with extendible hashing with single block buckets
(see binary numbers below)



In memory:

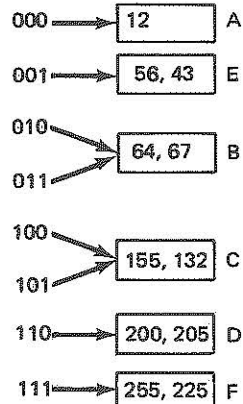
00	01	10	11
00 A	00 B	00 C	00 D
01 A	01 B	01 C	01 D
10 A	10 B	10 C	10 D
11 A	11 B	11 C	11 D

2. Now add 12.



00	01	10	11
00 A	00 B	00 C	00 D
01 A	01 B	01 C	01 D
10 E	10 B	10 C	10 D
11 E	11 B	11 C	11 D

3. Now add 205, 225, and 132.



00	01	10	11
00 A	00 B	00 C	00 D
01 A	01 B	01 C	01 D
10 E	10 B	10 C	10 F
11 E	11 B	11 C	11 F

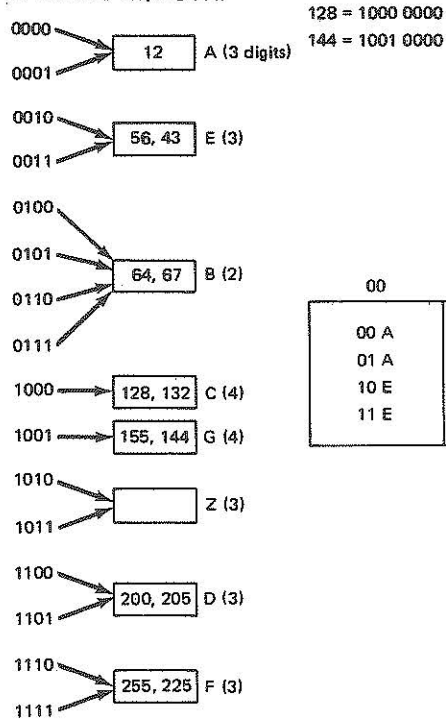
Numbers:

12 = 0000 1100	56 = 0011 1000
205 = 1100 1101	64 = 0100 0000
225 = 1110 0001	155 = 1001 1011
132 = 1000 0100	67 = 0100 0011
255 = 1111 1111	43 = 0010 1011
200 = 1100 1000	

Answers

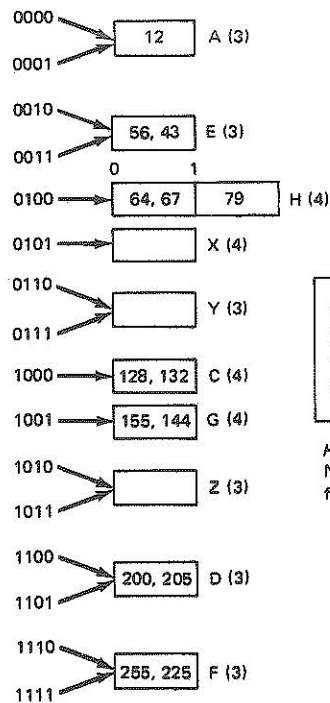
251

4. Now add 128, and 144.



00	01	10	11
00 A	00 B	00 C	00 D
01 A	01 B	01 C	01 D
10 E	10 B	10 Z	10 F
11 E	11 B	11 Z	11 F

5. Now add 79 (79, 64, and 67 all start with 0100). 79 = 0100 1111
We keep the number of digits with the bucket. Up to this point, this is just extendible hashing.



Add a size in index:

00	0A	00	1H	00	0C	00	0D
01	0A	01	0X	01	0G	01	0D
10	0E	10	0Y	10	0Z	10	0F
11	0E	11	0Y	11	0Z	11	0F

H has size 2^1 ; all others have size 2^0 .
Note that new space on disk is found for H.